A Novel Distributed Add-on Concept to Detect and Recover from Bus Failures in Controller Area Network using REDCAN

Håkan Sivencrona, Chalmers University of Technology
Göran Nilsson, Sauer-Danfoss
Fredrik Björn, Sauer-Danfoss

Abstract

In this paper a novel wire concept is presented that increases the dependability of a distributed communication system in a major way. It tolerates and can still provide full functionality in presence of one single fault of several types, such as short/open circuit, nodes that fail uncontrolled, so-called babbling idiots and other bus failures.
The concept utilizes a bus structure where the wire is divided into a plurality of sections. These sections are then interconnected to form an annular unit, ring, by means of a special module, REDCAN module. Each module comprises of a relay unit and logic, by which these can be coordinated with other circuits with the same functionality along the wire to define the left and right end of the wire to form a real broadcast bus. This is done by the individual circuit that decides to terminate the wire, one or two ends, or not at all – a transparent mode.
The paper presents two fault recovery techniques. The first is currently implemented in a CAN *kingdom* system, where the master controls the set-up of the broadcast bus. However, an algorithm that can be utilized by a fully distributed system e.g. CAN, is presented. The concept is furthermore intended to be protocol independent, and therefore the system should be useful for many existing embedded communication systems.

1 Introduction

The communication between embedded computers that controls applications with high requirements must be robust and reliable. To reach this state, dependability-increasing means is usually implemented in the system at several layers/levels in the system, including the physical level. These techniques include hardware and software and architecture so that the system can tolerate, handle and detect faults, errors and failures [1].
The faults in a communication system manifests in different ways but could be categorized as hardware and software faults.
Parts of the hardware are the chosen transmission media, driver circuits and redundancy. This together with the communication protocol as well as topology affects the possibility to meet set requirements [2].

All electric buses are for example sensitive towards open and short circuits, typical hardware faults. A short circuit can jeopardize the whole communication and seriously damage connected computer nodes. An open circuit on the other hand can degrade the system and cause the system to loose functionality in some parts of the system.
A software fault could be that erroneous data in a node could results in uncontrollably sending in either at the wrong time or as in a CAN system block other by continuously sending and thus ruin other's communication - a potential single point of failure.

A broadcast bus is hard to make fail silent, but on the other hand, topologies that rely on other protocols and methods are not always a better choice. Optical point-to-point communication, for example, has other fault models and drawbacks.
Given all pros and cons, the broadcast bus is still common and believed to be a very good solution for embedded control systems, due to for example low price, non-complex structure, etc.

Many concepts have emerged that aim to increase the reliability of the broadcast bus. Both hardware and software mechanisms have been implemented with different objectives and result.
A common strategy is the use of redundant channels, buses where the double bus makes it possible to tolerate single faults, both transient and permanent on one bus.
An extra step is the use of so-called bus-guardians. These only allow the guarded node to send on its own scheduled time thus protecting protects against timing faults, e.g. so-called babbling idiots [3]. This technique works well in a synchronous system. In an event triggered systems it is much harder to achieve a temporal firewall.
In the value domain many more mechanisms/features are known that can be implemented on different levels. These methods include features such as checksums and watchdogs as well as membership protocols that are used by other entities in the system to prevent the faulty node from sending incorrect data.

The problem often comes back to, if the nodes in the cluster can identify the faulty node in the system, which is not easy on a broadcast bus where the signal could be sent by anyone. It requires functions to exclude the faulty from the rest.
One strategy to do this is if the bus architecture could be re-designed run-time, with respect to signal path where the faulty node becomes disconnected.
It is very important that all nodes agree on such decision and that all functioning nodes avoid partitioning of the system.
A solution to these problems is proposed in this article.

REDCAN is a novel protocol and hardware that enables the coupling of several bus sections into a “broadcast” bus, i.e. all connected nodes see the same signal without delay and are connected electrically.
The papers is organized as follows; section two describes the functionality and hardware that has to be implemented in the communication controller to take care of the coupling of the sections to a broadcast bus. Section three describes the existing implementation and its functions, start up/restart in presence of faults as well as fault detection and establishing of a new signal path when fault occurs. Section four describes a distributed algorithm and the changes that must e implemented to set up the system utilizing the same REDCAN module, a fully distributed solution for a CAN system [4] and the new situation connected to such an architecture. Section six presents some thoughts concerning future research in order to improve the performance. Section number seven concludes the paper with discussion and conclusions.

2 The Add-on concept

The Add-on concept comes from the possibility to modify an existing broadcast bus system. The modification is done through dividing the bus into a feasible number of sections and then remove them from the drivers. The loose wires are then connected to a special module, the Redundant Can Module, RCM. The RCM is implemented in the node as a transparent interface between either the communication drivers or the bus or as a modified communication controller, CC, see figure 1.

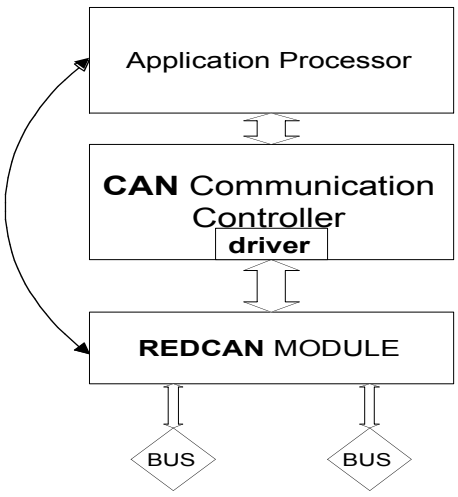


Figure 1 The REDCAN module in a CAN hierarchy.

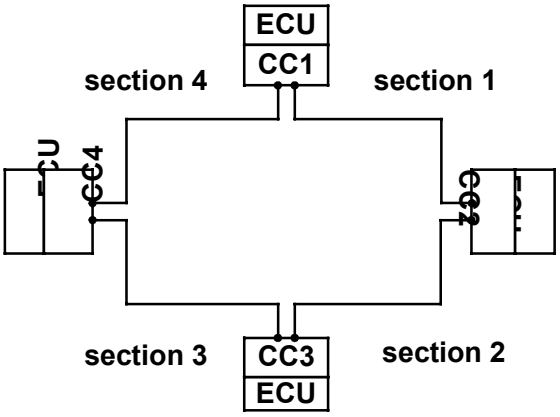


Figure 2 The bus structure with bus sections

The sections seen in figure 2, can host more than the two end nodes, but these communication controllers are not equipped with any RCM.
The RCMs shall define the left and right end termination of the bus wire. Thus, the individual RCM must choose to terminate left/right or not at all, a transparent mode to open up a signal path that in a fault free environment starts and ends in the same node. The transparent mode is a node where the termination resistance is disabled for the moment.
The application processor controls the functionality through combining two bits in the module.
The most critical task of the system configuration sequence is to manage the RCMs, and make them set their relays correctly. Three scenarios are handled, namely: during normal start-up or start-up in presence of faults as well as after detected errors.
To be able to know what decision to make, during start up or when a fault occurs, special knowledge is required. Every host must have information about transmit- and receive-times for the throughput of messages acknowledgement times of sent messages in the system. This is required to be able to activate the RCM mechanism correctly.
The status of the bus in run time must be checked by either heartbeat messages or by usage of information extracted from normal message frames.
The features of the system allow that faults can be detected in several ways. It can detect that a message that was expected never arrived, timing, omission and crash faults. As well as nodes that send incorrect data.

3 Centralized implementation

CANkingdom [5] is a special high-level protocol utilizing the CAN protocol. It has one master, the King, and the slaves that are called cities, controlled by the King, see figure 3. This hierarchy suits the centralized implementation very well. But some extra information is needed. The system needs the knowledge of the total number of RCMs to be able to decide about consistency in the system, e.g. no partitioning.

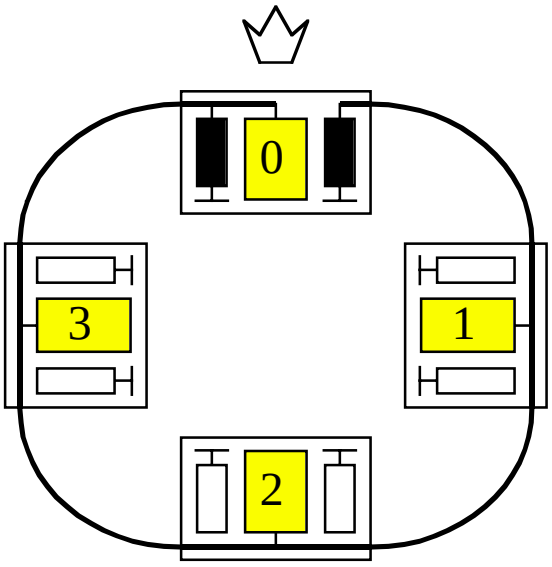


Figure 3 A CANkingdom architecture

The existing implementation utilizes a heartbeat protocol for detection. One node is appointed the master and this node transmits heartbeat messages with the interval T1. All the other nodes must respond to this within time T1. If the master does not work, another node will take care of the responsibility of being the master.
The master is designed so that if a node fails to respond one request, the master will decide this node faulty, e.g. if the node does not respond within 2*T1. When the master has detected the fault it ceases to send its heartbeats messages.
The slave nodes tolerate missing of one heart-beat message, that is, the slave node

will only act if it hasn't received any heart-beat during the time 2*T1.
The longest time for a system like this from fault occurrence to that the last node has detected the fault is, in the case of the master notifies some but not all nodes. 2*T1 after the fault arise the master cease to transmit the heartbeats. Another 2*T1 the nodes that did answer the last heartbeat will realize the occurrence of a fault. The longest time for fault detection is thus 4*T1.
In current implementation this time, T1 is 30 ms.
The fault detection mechanism can be activated by several reasons, for example at booting the system, in presence of a fault or to check if the fault was transient or permanent.
There are currently two algorithms for start up, fault detection and fault recovery for different high-level protocol. One method utilizes on randomly testing right/left while the implemented method utilizes the master (CANkingdom) in the system to make a sequential scan of each node, first right and then left. This method requires that all nodes are starting up approximately simultaneously.
When the sequence starts all nodes, except the master, connect left, see figure 4.

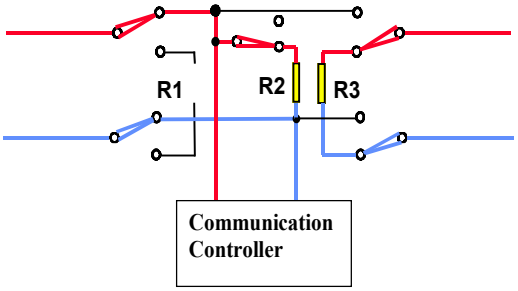
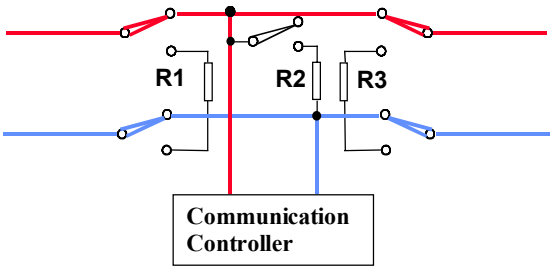


Figure 4. Connected left, disconnected right

These slave nodes then wait for a master message. When this is received a flag is set in the slave node that left connection is functioning.
The node then enters the transparent mode; see figure 5, and then waits for the master.
The slave node then also receives a message from the master, stating that the right connection was okay.



Fi
gure 5. Circuit in transparent mode, default mode

After a time T2, the slave node will take a decision following table 1.

Left OK	Right OK	Decision
True	True	Stay in transparent m
True	False	Connect left, fault det
False	False	Connect right, await r

Table 1 Decision table for slave node

If the node was allowed to turn right the fault detection is ready whenever the master contacts the node.
The master starts up the fault detection sequence by connecting right. It polls the first node to the right. This knowledge is defined in the system design. If this turns out okay the slave node will set transparent mode and the master may connect to next node. If this is also going well the master will tell the first node that the nodes right neighbor is okay as well. This continues until a node does not respond. If the node can connect to the last node the master will test left connection. If that is true the fault detection is ready and the master hold both terminations.
If the master cannot connect with all nodes within T2 the master will turn left and make the sequential testing in the same way as before, just opposite direction.
If not all nodes have been reached within another T2 from the time the master connected left and the nodes that did not respond are assumed disconnected.
T2 must be chosen so that the master can reach all nodes during this time. Parameters that affect this are for example the baud rate and the total number of nodes. It is important to realize that all nodes do not start their

clocks at the same time. This depends on the detection of fault at different times.
In the existing implementation, a CAN system, T2 is chosen to be 500 ms, the number of nodes have varied between 3 and 10. No tests have been done so far to optimize T1 and T2. The time for recovery is estimated to be at maximum 1 s.

4 Distributed algorithm

The centralized CAN kingdom algorithm is very specific and suited for that protocol and thus not optimized for other protocols. A fully distributed system is believed to have higher performance and can be more resilient towards faults in the system.
A node can enter four Modes; Scan = The node randomly tests to left and right to see if it can connect to a neighbor, Connect = The node has contact to one neighbor and tries to connect to the other, ConnectScan = The node have contact with both neighbors but is not yet allowed to go into transparent mode and finally Transparent = The node have contact with both neighbors and is connected transparent.
The scan mode is a random function that will either try to transmit set-up messages left or right with information such as identity and other information that it necessary to establish a system. If no connection is established it will wait for a random time. This time is multiplied with the time it takes for one frame to be received/transmitted. The time varies for example between 4 and 8 frame times, and then the direction (mode) is changed. During this time the node listens through the whole time but one frame time when it transmits its data. The node has furthermore a unique node number that can be the same as used by the host.
A message to a Neighbor node includes four parameters, O = Own node number, N = Neighbor node number (only if known), S = if Own Scan Mode is on, T = if allow neighbor to enter transparent mode/Neighbor allows us to enter transparent mode. There is furthermore another message type that asks for a specific node, D? = Asks if nearest neighbor on the not connected side is present, O? = How this message is understood at the receiver. A node, which is in transparent mode, can respond to an N? message with O! = Node number O is present
Upon start up of the nodes the random scan is activated, see figure 7.

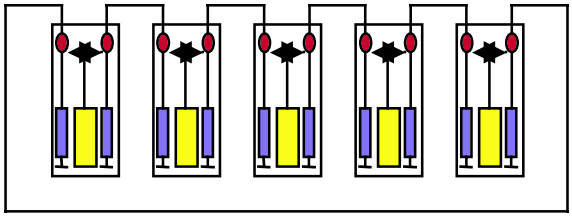


Figure 6 All nodes in scan mode, A to E

When any node (A) hears a message it is allowed to answer it answers, and depending on the current mode it acts accordingly. When the connection has been acknowledged both nodes (A and B) will act either by entering the scan mode again or enter a transparent mode. This depends if they have established a connection in opposite direction before (E, C). The node that transmits first (B) and also has a known connector to the opposite direction may enter the transparent mode, if the node is not the same, as it already knew. The node (B) must therefore first send one message in the opposite direction and ask for the node it negotiated with (B) to protect against a non-terminated system.
When the node (A) has entered scan mode again it tries to connect to another node (C) in the old direction (over B). If no response the node can time-out and consider the system in a degraded mode. Or the system is established, see figure 8. For pseudo code, see table 2.

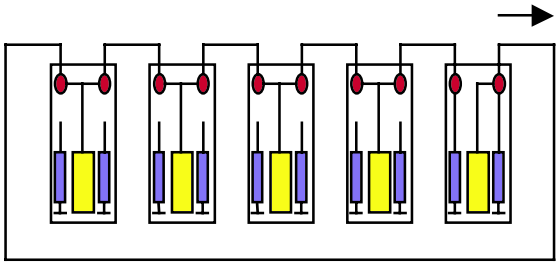


Figure 7 Request send to see if transparent mode is allowed

In Scan mode, Connect mode and ConnectScan mode the node have a Direction, (Dir), Left = The node is connected to the left or Right = The node is connected to the right.
In Scan Mode the node can have two states, NoResponseYet = Waiting for message from neighbor and WaitForSecondResponse = Waiting for message from neighbor.
In Connect Mode the node can have two states, NoResponseYet = Waiting for message from neighbor and WaitForSecondResponse = Waiting for message from neighbor.
In ConnectScanMode the node can have two states, NoResponseYet = Waiting for message from neighbor and WaitForSecondResponse = Waiting for message from neighbor.

```
var
  Ready          := False;
  OK             := False;
  Mode           := Scan;
  ScanState      := NoResponseYet;
  ConnectState   := NoResponseYet;
  ConnectScanState := NoResponseYet;
  Dir            := Random(Left,Right);
while not Ready do
begin
  case Mode of
    Scan :
    begin
      case ScanState of
        NoResponseYet :
        begin
          if not Rx then
          begin
            // If Time out change direction
            if RandomTimeOut then
              ShiftDirection(Dir);
            Tx( O S ); // Own Addr and Scan mode
          end
          else // Message from neighbor
          begin
            case Rx of
              N  : begin // Neighbor addr
                     Tx( O N S T );
                     ScanState := WaitForSecondResponse;
                   end;
              NO : begin    // Neighbor and Own addr
                     Tx( O N S T );
                     Mode := Connect;
                     ShiftDirection(Dir);
                   end;
              NS : begin // Neighbor addr and Scan mode
                     Tx( O N S )
                     ScanState := WaitForSecondResponse;
                   end;
              NOS: begin     // Neighbor and Own addr and Scan mode
                     Tx( O N S );
                     Mode := Connect;
                     ShiftDirection(Dir);
                   end;
            end; // case Rx
          end;
        end; // ScanState NoResponseYet;
        WaitForSecondResponse :
        begin
          case Rx of
            NO : begin // Neighbor and Own addr
                   Mode := Connect;
                   ShiftDirection(Dir);
                 end;
          end; // case Rx
        end; // ScanState WaitForSecondResponse
      end; // case ScanState
    end; // State Scan
    Connect:
    begin
      case ConnectState of
        NoResponseYet :
        begin
          if not Rx then
            Tx( O ); // Own Addr
          else
          begin
            case Rx of
              N  : begin // Neighbor addr
                     Tx( O N T );
                     ConnectState := WaitForSecondResponse;
                   end;
              NS : begin // Neighbor addr and Scan mode
                     Tx( O N );
                     ConnectState := WaitForSecondResponse;
                   end;
              NO : begin    // Neighbor and Own addr
                     Mode := ConnectScan;
                   end;
              NOT,         // Neighbor and Own Addr and
Transparent OK
               NOST:begin // Neighbor and Own Addr and Scan mode
and Transparent OK
                     Tx( O N );
                     Mode = Transparent;
                     SetTransparentNode();
                   end;
            end; // case Rx
          end; // if Rx
        end; // ConnectState NoResponseYet
        WaitForSecondResponse :
        begin
          case Rx of
            NO : begin // Neighbor and Own addr
                   Mode := ConnectScan;
                 end;
            NOST:begin // Neighbor and Own Addr and Scan mode and
Transparent OK
                   Mode = Transparent;
                   SetTransparentNode();
                 end;
          end; // case Rx
        end; // ConnectState WaitForSecondResponse
      end; // case ConnectState
    end; // Mode Connect
    ConnectScan :
    begin
      case ConnectScanState of
        NoResponseYet :
        begin
          if not Rx then
          begin
            // If Time out change direction
            if RandomTimeOut then
              ShiftDirection(Dir);
            // Ask for the nearest node on the other (not Dir) side
            Tx( D?)
            // And ask for nearest not transparent neighbor
            Tx( O );
          end
          else // Message from neighbor
          begin
            case Rx of
              N   :begin // Neighbor addr
                     Tx( O N T );
                     ConnectScanState := WaitForSecondResponse;
                   end;
               NOT :begin // Neighbor and Own addr and Transparent
OK
                     Tx( O N ) ;
                     Mode = Transparent;
```

```
                    SetTransparentNode();
                  end;
               D!  :begin    // Response from nearest node on the
other ( not Dir ) side
                     Ready := True;
                   OK := True;
                  end;
             end; // case Rx
           end;
         end; // ScanState NoResponseYet;
         WaitForSecondResponse :
         begin
           if Rx = NO then // Neighbor and Own addr
             ConnectScanState := NoResponseYet; // Keep looking for
nodes
         end; // ScanState WaitForSecondResponse
       end; // case ConnectScanState
    end; // State ConnectScan :
    Transparent:
    begin
      if Rx = O? then // Request for this specific node
        Tx( O! );
    end; // Mode Transparent
  end; // case Mode
  Ready := ReadyTimeOut;
end;   // while not ready
// If Time out
if not OK then
begin
  case Mode of
    Scan       : // Whatever is best if no neighbor found ( >1 error )
    Connect    : ShiftDirection(Dir);    // 1 neighbor found
    ConnectScan : begin
                  SetTransparentNode(); // both neighbors found
                  OK := True;
                end;
    Transparent : OK := True;                // both neighbors found
  end
end;
```

Table 2 The algorithm in pseudo code

It is furthermore possible to improve the algorithm by letting the node that first notify an established system transmit OK.

5 Possible implementations

One possible implementation of a distributed REDCAN system is to modify the CAN controller. The controller could then be totally transparent from the host point of view. It would consist of logic that allows the scanning of messages on the bus and act due this information. It would manage the two relays to establish the broadcast bus, see figure 8.

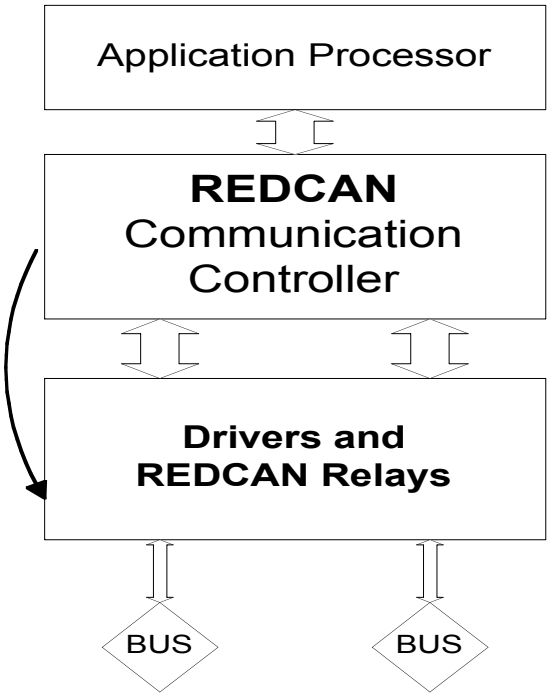


Figure 8 An implementation with a specifically designed Communication Controller, CC

The Controller could act on three different cases. The first is to have own identifiers for the REDCAN communication, i.e. messages that are only sent when there is a need and with low priority. This could be solved through some dynamic scheduling [6]. Second is to piggyback normal messages sent from the host and finally produce non-formatted messages that would be used by the REDCAN functionality but discarded by the application.

6 Future work

Fast fault recovery is an important property of a communication system. The centralized algorithm has not been designed with this in mind and couldn't provide short recovery times, thus the introduction of a distributed concept.
One of the first tasks will be to validate the algorithm, with for example formal methods.
When the algorithm has been implemented in a CAN system, it is the scoop of the authors to test and verify the behavior and optimize the start-up algorithms that have been proposed. The function could hopefully be implemented in a block transparent from the

host and system. There is also possible to improve this algorithm by means of minimize the message overhead.

7 Conclusions

The REDCAN concept is a suitable concept for harsh conditions and can, although the bus is incommunicable due to babbling idiots or short circuits, find new paths for the messages and also disconnect the failing node. The centralized algorithm can detect and recover from the same fault types as this, for example short cuts and babbling idiots. Failures such as one faulty node can always be disconnected but more than one bus failure can be detected although the communication might be degraded. The most significant difference between them is that a distributed algorithm does not have a master that jeopardizes the whole system start-up. Compared to the centralized it is more useful for general CAN systems that are fully distributed.

The performance has yet not been fully verified but the time from a missing message or occurrence of a fault to the time for the communication to run again is in the far from optimized (centralized) system around 500 ms. With a fully distributed system it is not unrealistic to aim for times in the area of maybe two communication cycles, e.g. nodes have sent two times, this remains to be tested.
With the sequential fault detection algorithm the time to recover from a fault is dependent on the amount for nodes in the system. The random method algorithm is not so heavily depending on the number of nodes. However this is an interesting field to investigate and compare different algorithms.

8 References

[1] J.C. Laprie, *Dependability – Its Attributes, Impairments and Means:* Springer-Verlag, 1995.

[2] Christian, F., "*Understanding Fault Tolerant Distributed Systems*", Communication of the ACM, 1991.

[3] Temple C., "*Avoiding the babbling-idiot failure in a time-triggered communication system*", Fault-Tolerant Computing, 1998. Digest of Papers, Twenty-Eighth Annual International Symposium on, 1998.

[4] CAN, SS-ISO 11898, Road Vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication.

[5] CAN Kingdom 3.01 specification

[6] K. M. Zuberi and Kang G. Shin, "*Non-Preemptive Scheduling of Messages on controller Area Network for Real-Time Control Applications*", Copyright IEEE 1994.

[7] Jalote P., *Fault Tolerance in Distributed Systems*. (Prentice Hall, NJ, 1994).

[8] Ö. Askerdal, *Fault detection and handling:* PALBUS10: 5, 2000.

Chalmers University of Technology
Department of Computer Engineering
SE 412 96 Göteborg
+46(0)31-772 1669
+46(0)31-772 3663
E-mail: sivis@ce.chalmers.se
Website: www.ce.chalmers.se

Sauer-Danfoss
Stålgatan 1
SE-343 34 Älmhult
+46 (0) 476-56918
+46 (0) 476-56944
E-mail: fbjorn@sauer-danfoss.com
gnilsson@sauer-danfoss.com
Website: www.sauer-danfoss.com