First prototype and experimental assessment of media management in ReCANcentrate

Manuel Barranco, David Geßner, Julián Proenza
Dpt. Matemàtiques i Informàtica
Universitat de les Illes Balears, Spain
manuel.barranco@uib.es

Luís Almeida
IEETA / DEEC-FEUP
Universidade do Porto, Portugal
lda@fe.up.pt

Abstract

Although the use of star topologies to improve dependability in field-buses is gaining in importance, as in TTP/C and FlexRay, a mature technology such as the Controller Area Network (CAN) remained essentially a bus-only network. Thus, we proposed a CAN-compliant replicated star topology called ReCANcentrate, which has advanced error-containment and fault-tolerance mechanisms. Its two hubs are coupled with each other and create a single logical broadcast domain that allowed us to propose, in a previous work, a strategy for each node to easily manage the replicated star by means of a software driver that abstracts away the details of the replication. This paper describes the main functionalities of this driver, as well as the first tests we have conducted, on a real ReCANcentrate prototype, to verify the correctness and the performance of the driver in the absence and in the presence of faults.

1 Introduction

The use of star topologies as the underlying topology of field-bus communication subsystems has been a matter of main concern, given the stars potential dependability benefits. This can be clearly seen in recent protocols such as TTP/C [1] and FlexRay [2]. However, the Controller Area Network [3] (CAN) protocol, which is one of the most mature field-bus technologies, remained essentially a bus-only network. Thus, in order to benefit from stars and, in particular, from their better error-containment and fault-tolerance capabilities, we proposed a CAN-compliant replicated star topology called ReCANcentrate [4], whose significant reliability benefits have been quantitatively corroborated recently [5].

The basic architecture of ReCANcentrate is depicted in Figure 1. It includes two hubs with nodes connected to each of them by dedicated links containing each an uplink and a downlink. Both hubs are also interconnected by at least two interlinks, each containing two independent sublinks, one for each direction. Each hub has mechanisms to contain errors originated at nodes, links, interlinks, or hubs [4]. ReCANcentrate also has mechanisms to tolerate faults occurring at one of the hubs, at links/interlinks, and at the nodes' communication controllers.

To achieve fault tolerance, each ReCANcentrate star is used as a CAN channel that transmits a replica of the same

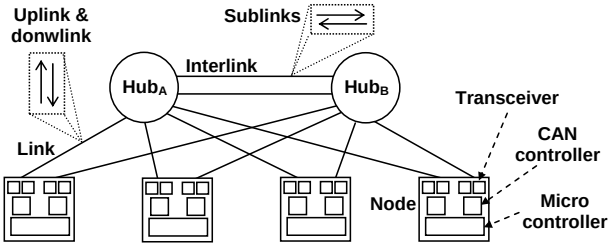

Figure 1. ReCANcentrate architecture

data in parallel. However, due to CAN's error-signaling and arbitration mechanisms, a bit error in one channel is enough for its traffic to evolve different than in the other replica. Thus, it is not easy for a node to detect when frames received at different instants of time, each through a different channel, are copies of the same frame (duplicates); or when a frame received from one channel is omitted from the other (omissions). Moreover, since channels are independent, the network can become partitioned if faults prevent nodes from communicating through different replicas [4]. To overcome these problems, the hubs exchange their traffic through the interlinks and couple with each other [4], thereby synchronizing both channels at bit level and forcing both hubs to quasi-simultaneously transmit the same value, bit by bit, through their downlinks. Note that, thanks to the coupling between the hubs, all nodes can communicate with each other regardless the hub or hubs each node is able to communicate through.

The above mentioned features allowed us to define a strategy for each node to easily manage transmissions and receptions, and to treat faults occurring in the stars [6]. Moreover, in a previous work [7] we presented the ongoing development of a software driver that executes on each ReCANcentrate node and that implements this management. The current paper completes previous work by describing the main routines that constitute this driver, as well as by explaining the results of the first tests we have conducted, on a real ReCANcentrate prototype, in order to verify the correctness and the performance of the driver in the absence and in the presence of faults.

2 Media management basics

The physical layer of CAN implements a wired-AND function of every node contribution, thereby providing the *dominant/recessive transmission* property [3], which ensures that a dominant bit, '0', prevails over a recessive bit,

 doi:10.1109/ETFA.2010.5641321

'1'. Additionally, the CAN bit synchronization guarantees the *in-bit response* property, thanks to which nodes quasi-simultaneously observe every single bit on the channel.

The ReCANcentrate hubs perform a special AND-coupling within a fraction of the bit time, thereby creating a single logical broadcast domain that keeps the two referred CAN properties; thus, both hubs behave like one, transmitting the same value bit by bit in their downlinks. In order to connect to the replicated star the node relies on the architecture depicted in Figure 1. It is constituted by commercial-off-the-shelf (COTS) components only: one microcontroller and two CAN controllers, each connected to a separate hub, using one transceiver for the uplink and another one for the downlink.

2.1 Management in the absence of faults

According to our media management strategy, the node transmits towards one of the hubs only, while receiving from both hubs at the same time. One controller, the *non-transmission controller* (*non-tx controller*), is only used to receive frames—which may have been transmitted by its own node or by other nodes. The other controller, the *transmission controller* (*tx controller*), is used by the node to transmit frames in addition to receiving frames.

When a frame is successfully exchanged through the network, i.e. when a *delivery event* occurs, each node expects that its two controllers quasi-simultaneously notify of that event. Thus, in the absence of faults, the node manages transmissions and receptions as follows. First, if the node successfully transmits a frame, the *tx controller* and the *non-tx controller* notify of the transmission and reception of this frame respectively; thus, the node only needs to accept the notification of the transmission as valid and release the reception buffer of the *non-tx controller*. Second, if the node receives a frame sent from another node, it is notified of this reception by its two CAN controllers. When this happens, the node must merely consume the frame received at one of the controllers and, then, release the reception buffers of both controllers.

2.2 Management in the presence of faults

ReCANcentrate's fault model includes faults at nodes, links, interlinks, or a hub that manifest themselves, from a channel point of view, as stuck-at or bit-flipping streams [4]. Additionally, it includes CAN controller faults that manifest as a *crash* from a node's point of view, i.e. faults that lead a CAN controller to notify nothing to its node. The only fault assumption is that hubs remain coupled with each other using at least one non-faulty interlink.

Channel errors generated by a fault block the communication in both stars as long as the hubs do not isolate it by disabling the appropriate hub ports [7]. Once isolated, if the fault affects a link or a CAN controller, it only prevents the corresponding node from communicating through the corresponding hub; however, if the fault affects a hub, it can lead to no node being able to communicate through that hub. Interlink faults do not prevent any node from communicating, as long as they do not affect all interlinks.

To tolerate a fault, it is necessary that a node that cannot communicate through a given hub as a consequence of that fault continues to communicate through the other hub. As explained in [6], a node that cannot communicate through a given hub will observe a *notification omission discrepancy* (omission discrepancy for short): when a *delivery event* occurs, the node observes that the controller connected to that hub fails to notify that event. Thus, in principle, the node can tolerate a fault by simply accepting as valid the transmission/reception notified by the controller that has no problems. Note that if the controller that omits notifications is the *non-tx* controller, the node does not need to diagnose it as faulty. However, if the controller that omits is the *tx controller*, the node must eventually diagnose it as faulty and rule it out for communicating; otherwise, the node will not be able to transmit anymore. Thus, the node initiates a *transmission timer* (*tx timer*) when it requests a transmission: if the timer expires before the *tx controller* notifies of a successful transmission, the node rules it out and uses the other controller to transmit/receive. Additionally, to enhance the node's fault diagnosis capabilities, we propose to rule out a CAN controller whenever its *Transmission Error Counter* (TEC) or its *Reception Error Counter* (REC) [3] reaches a given threshold—this prevents controllers from going into the *error-passive state*, in which they could inconsistently exchange frames [3].

Finally, there are some situations in which an omission discrepancy can be caused by a media fault that does not prevent all controllers from communicating, but that leads them to inconsistently exchange a frame. On the one hand, this may happen in the presence of any of the error scenarios affecting the last-but-one bit of a frame that have been identified for CAN [8]. On the other hand, a stuck-at-recessive fault may provoke an inconsistency if it prevents a controller from monitoring the traffic, or if it affects its uplink and prevents it from aborting an on-going frame. For instance, if a downlink is stuck-at-recessive during the broadcast of a whole frame, the controller connected to that downlink will not receive it. The media management we propose takes into account the scenarios of [8] to some extent, as well as inconsistencies provoked by stuck-at-recessive faults; assuming that a frame is not inconsistently exchanged more than *maxIncon* consecutive times.

3 Driver architecture

We designed the media management to be implemented as a driver that abstracts away the details of the node architecture and the media replication. Figure 2 depicts the basic structure of the driver. It shows the peripherals the driver requires: two CAN controllers, *CAN 1* and *CAN 2* in Figure 2, and a timer to be used as the *tx timer*. Additional hardware requirements are the the ability to generate interrupts through software, interrupts that can be nested, interrupt generation when a controller's TEC/REC reaches a threshold, and configurable priorities for the interrupts.

At the top part of the structure we can see the interface the driver provides to the application, i.e. the *driver inter-*

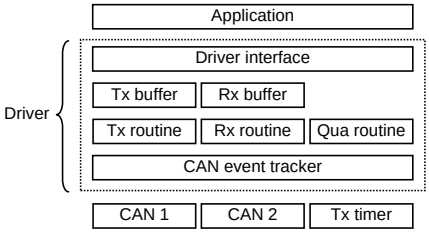

Figure 2. Basic driver structure

face. It includes a set of primitives that abstract away the existence of two CAN controllers and that allow the application to communicate through the replicated channel.

Below the interface, we can find the driver's *transmission buffer* (*tx buffer*) and *reception buffer* (*rx buffer*). When the application requests to transmit a frame, the driver not only writes that frame to the hardware transmission buffer of the *tx controller*, but it stores a copy of that frame in the driver's *tx buffer*. The driver needs this copy for different management operations, e.g. if the driver diagnoses the *tx controller* as faulty before that controller successfully transmits the requested frame, the driver automatically transfers a copy of that frame to the surviving controller. Regarding the *rx buffer*, it is a buffer that accommodates the last frame received through ReCANcentrate. When the driver accepts a frame reception, it immediately copies the frame from the hardware reception buffer of one of the controllers to the driver's rx buffer and releases the hardware reception buffers of both controllers.

The major part of the driver functionality is located in the *management routines*: the *transmission routine* (*tx routine*), the *reception routine* (*rx routine*), and the *quarantine routine* (*qua routine*). Each one of them is an interrupt service routine (ISR) that handles a given CAN controller or timer notification. The *tx routine* and the *rx routine* are executed when any of the two CAN controllers notifies of a transmission and a reception respectively. The *qua routine* is executed when the TEC/REC of any of the two CAN controllers reaches a specific threshold or when the *tx timer* expires. To simplify the routines we considered that they cannot be nested, which requires that all of them have the same execution priority.

However, none of these ISRs is directly triggered when a notification occurs. Instead, what a notification triggers is another ISR called *CAN event tracker*. This ISR has the maximum execution priority so that it can preempt any management routine. When a notification occurs, the CAN event tracker annotates that it has occurred and, then, triggers the execution of the appropriate management routine by generating an interrupt. The triggered management routines will be pending until the CAN event tracker and any previously preempted management routine end. If both a qua routine and a rx routine for the same controller are pending, the implementation must ensure that the qua routine is executed last because, as will be explained in the next section, the qua routine deactivates the controller and the rx routine would otherwise access the receive buffer of an deactivated controller.

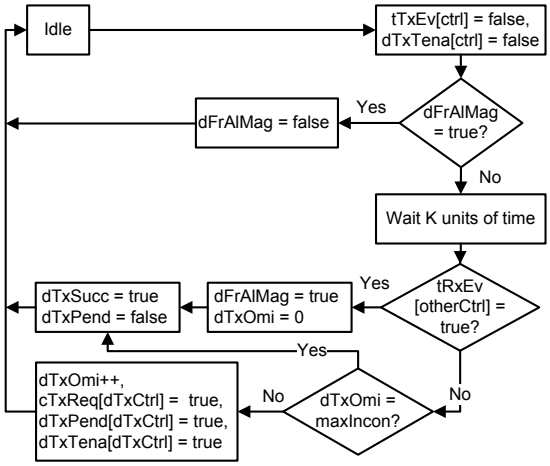

Figure 3. Tx routine

The *CAN event tracker* can be seen as a dispatcher that decides which routine must handle each notification. But its most important functionality is to annotate what notifications occur (and thus what routines it has triggered). For that, it uses a boolean variable for each type of notification, which we call *tracking variables*. They are needed to handle each *delivery event*, as explained later.

4 Management routines

4.1 Tx and Rx routines

When a *delivery event* occurs, it is expected that each CAN controller notifies about a transmission or reception by triggering, respectively, an execution of the *tx routine* or *rx routine* for each notification. If this happens, one of the routines will execute before the other, but both must collaborate to handle the event. In contrast, if only one routine is triggered because a CAN controller omits due to a fault, that routine must handle the event alone.

Next we briefly explain the actions carried out by the tx routine, which is depicted in Figure 3. First, the tx routine resets the tracking variable that indicates that it has been triggered by the corresponding controller (*tTxEv[ctrl]*). It also resets the tx timer associated with the transmitted frame. Then, since a frame transmitted by the tx controller should be received by the non-tx controller, the routine checks if the rx routine was triggered first and has validated the correspondence between the frame transmitted and received. For this purpose, it consults the driver variable *dFrAlMag*. If its value is true, it means that the rx routine has already managed the transmission notification and, thus, the tx routine simply needs to reset this variable to false. If *dFrAlMag* is false, the routine waits K units of time to give enough time to the non-tx controller to notify the reception of the transmitted frame. Afterwards, it consults the tracking variable *tRxEv* to elucidate if this notification has occurred. If so, the routine sets *dFrAlMag* to *true* to inform the rx routine that it has validated the correspondence between the notifications, so that the rx routine does not have to check it again. Then, it resets the driver variable *dTxOmi*, whose role will be explained later on.

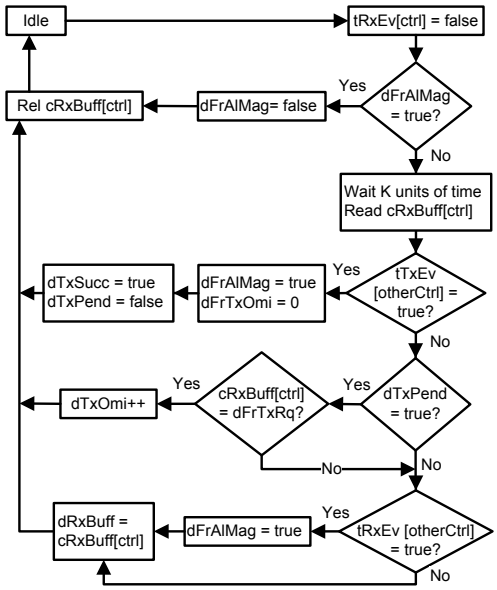

Figure 4. Rx routine

Finally, it indicates to the application that the frame has been successfully transmitted (*dTxSucc = true*) and resets the driver variable *dTxPend*, which indicated that the frame was pending to be transmitted.

In contrast, if after waiting K units of time, the rx routine has not been triggered, the tx routine detects an *omission discrepancy* (Section 2.2). When this happens, it checks if the number of omissions detected so far is equal to the maximum number of consecutive inconsistencies: *maxIncon*. If so, the tx routine assumes that this inconsistency is due to a permanent stuck-at-recessive fault that prevents the non-tx controller from communicating. Thus, it simply indicates to the application that the frame has been successfully transmitted. Otherwise, it increases the omission counter *dTxOmi* and requests again the transmission of the frame through the tx controller (*dTxCtrl*). Note that this re-transmission strategy leads the node to generate duplicated frames in case the omission discrepancy was actually due to a permanent stuck-at-recessive fault.

The rx routine is similar to the tx routine (see Figure 4). One of the first actions it carries out is to check the driver variable *dFrAlMag*. If it is true, it means that the frame whose reception has launched the rx routine has been already managed by the tx/rx routine triggered by the other controller; thus, the rx routine merely resets this variable and releases the receive buffer of its corresponding controller. Otherwise, the rx routine knows that it is the one that is executing first, thus it waits K units of time (to give the other controller time to notify) and reads the frame received at the receive buffer of its corresponding controller. Then, the rx routine checks whether or not the other controller has triggered a tx routine. If affirmative, it means that the frame the rx routine is managing is actually a frame transmitted by the other controller. In that case, the rx routine sets *dFrAlMag* to *true* to inform the tx routine—which will be executed next—that it already managed the transmission notification. Next, the rx routine resets the

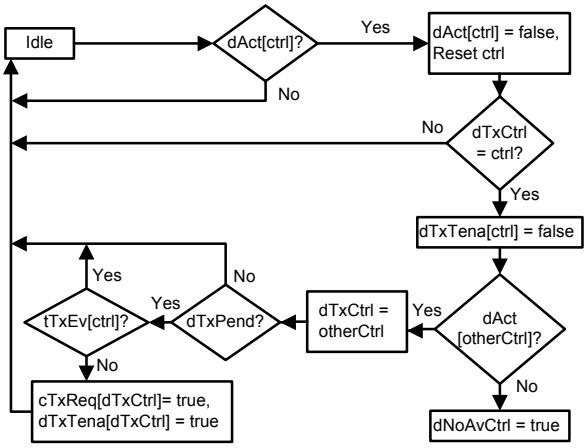

Figure 5. Qua routine

driver variable *dTxOmi*, indicates to the application that the frame has been successfully transmitted, and resets the driver variable *dTxPend*. In case the other controller has not triggered a tx routine when the rx routine has already waited K units of time, the rx routine still needs to elucidate if it is managing a frame transmitted by that other controller. This is so because the other controller could omit the transmission notification due to a fault, e.g. it could crash before triggering the tx routine; or a CAN inconsistency scenario could lead it to believe that it did not successfully broadcast the frame. In order to make sure that it is not managing a frame transmitted by the other controller, the rx routine checks if the driver variable *dTxPend* is true and if the frame placed at the tx buffer of the driver (*dFrTxRq*) is equal to the frame it has read from its corresponding controller (*cRxBuff[ctrl]*). If so, the rx routine assumes that an omission discrepancy occurred and, thus, increases the omission counter *dTxOmi*. Note that in case the omission is due to a permanent fault, e.g. the tx controller has crashed, then the tx timer will eventually expire and the qua routine will carry out the actions needed to tolerate the fault (see Section 4.2). Finally, in case the rx routine is not managing a frame transmitted by the other controller, it checks if that other controller has also received the frame and has triggered the corresponding rx routine. For this purpose, the rx routine consults the appropriate tracking variable (*tRxEv[otherCtrl]*). If it is true, it sets the driver variable *dFrAlMag* to true to indicate to the other rx routine that it has already managed the reception of the frame. In any case, at the end, the rx routine copies the received frame to the driver's reception buffer (*dRxBuff*) and releases its controller's reception buffer.

4.2 Qua routine

When a TEC/REC of a controller reaches a given threshold, it triggers the qua routine in order to diagnose the controller as faulty. Similarly, when the tx timer expires, the qua routine is also triggered in order to rule out the tx controller for communicating, as this expiration implies that this controller has crashed (see Section 2.2).

The diagram of this routine is depicted in Figure 5. Ini-

tially the routine sets the controller diagnosed as faulty to *non-active* (for this purpose it sets the driver's variable *dAct[ctrl]* to false) and resets it. Then, it elucidates whether or not the controller is the tx controller. If not, the routine merely finishes; otherwise, it has to perform a set of actions needed to start using the surviving controller as the new tx controller. If this is the case, the routine first disables the tx timer: this prevents the timer from unnecessarily expiring in case the qua routine is triggered because the tx controller encountered too many errors and the timer was still running. Afterwards, the qua routine checks if the other controller is non-faulty, i.e. if the other controller is *active* (*dAct[otherCtrl] = true*). If not, the routine indicates to the application that there is no controller left to communicate (*dNoAvCtrl* is set to true) and finishes. Otherwise, the routine marks the other controller as the tx controller (*dTxCtrl = otherCtrl*). Then, the routine has to elucidate whether or not a frame's transmission is pending because, in that case, it could be necessary to request the transmission of that frame through the new tx controller. Specifically, the qua routine requests this transmission if the driver variable that indicates that there is a pending transmission (*dTxPend*) is true and the tracker variable that indicates that a tx routine has been triggered (*tTxEv*) is false. Note that the qua routine does not request the transmission if the tx routine has been triggered. This is because the TEC/REC of the old tx controller may have reached the predefined threshold just after completing a transmission and, thus, to retransmit would generate a duplicate.

4.3 Example of execution

Figure 6 depicts an example scenario which involves the execution of some of the above-mentioned routines. The beginning of the chronogram represents the end of an 8-byte-data frame sent by another node. *CAN 1* and *CAN 2* refer respectively to the tx controller and the non-tx controller of a receiving node. The frame is observed at the CAN 1 and CAN 2 downlinks of that node. The CAN 1 uplink has been isolated by the corresponding hub due to prior errors (not shown). When the frame ends, each CAN controller triggers a CAN interrupt (represented by an upward arrow), which is captured by the corresponding CAN event tracker ISR. Additionally, the chronogram shows that the CAN 1 controller triggers a second CAN interrupt after its REC reaches a predefined threshold, known as error warning, due to an erroneous bit in its downlink during the third bit of the intermission period.

However, note that in the chronogram the CAN event tracker ISRs cannot execute immediately after their controllers notify a CAN event. This is so because these notifications take place just after the driver disables the micro's interrupts as a consequence of the execution of a transmission request primitive (*tx req*) called by the application—this primitive needs to disable the interrupts to ensure mutual exclusion when accessing driver variables such as the tx buffer. Once tx req finishes, the tracker ISRs execute: they discern what caused the CAN interrupt (a reception in case of CAN 2, and both a reception and an error warning in case of CAN 1), they annotate (by means of the tracking variables) that the discerned interrupts occurred, and they trigger, by means of software interrupts (represented as downward arrows), the corresponding management routines. The one that executes first is the rx routine of the tx controller (CAN 1): it basically reads the rx buffer of CAN 1, informs the application that a new frame has been received, and releases the receive buffer. This routine also knows, thanks to the appropriate tracker variable, that the rx routine for CAN 2 will execute next. Thus, when finishing, it sets *dFrAlMag* to *true* to inform the other rx routine that the reception was already managed. Then, when the rx routine of CAN 2 executes, it merely resets *dFrAlMag*, releases the rx buffer of CAN 2, and finishes.

Finally, the qua routine for CAN 1 executes. It deactivates CAN 1 and marks CAN 2 as the new tx controller. Since the application previously requested a transmission (see beginning of the chronogram) and the old tx controller failed after completing it, the qua routine also requests the pending transmission through the new tx controller.

5 Prototype and tests

We built a ReCANcentrate prototype provided with two hubs, two interlinks, and three nodes. Each hub is implemented using the VHSIC Hardware Description Language (VHDL) and synthesized in a Xilinx Spartan-3 XC3S1000 FPGA—except its hardware interface, which is built using COTS transceivers. One UTP ethernet cable and a pair of RJ45 connectors are used to implement each link/interlink, which includes an uplink and an independent downlink (or two independent sublinks). Each uplink, downlink, and sublink uses two-wire differential lines.

We implemented the media management driver for the dsPIC30F6014A [9]. Each node consists of two different boards attached to each other. The first one is a printed board called dsPICDEM™that Microchip™provides for evaluation purposes. It includes the dsPIC30F6014A microcontroller, which has two CAN controllers and which also provides the appropriate interrupt sources and allows to configure the priorities and the preemption policy of the ISRs as we need (Section 3). The dsPICDEM™board also includes a row of 4 LEDs the micro can use for debugging purposes. The second board is called *starLink* and we designed it specifically for ReCANcentrate; it basically includes COTS transceivers to connect the dsPIC30F6014A's CAN controllers to the replicated star.

During the development our focus was on making the source code modular, readable, and easy to modify, while following a *defensive programming* [10] approach by inserting *assertions* throughout the code, i.e. code that actively checks whether specific assumptions hold at a given point of a program. Despite increasing the node's overhead, this approach allowed us to easily debug the code as well as to verify the correctness of the driver's functions.

The objective of this driver is to demonstrate the feasibility of the media management we propose for ReCAN-

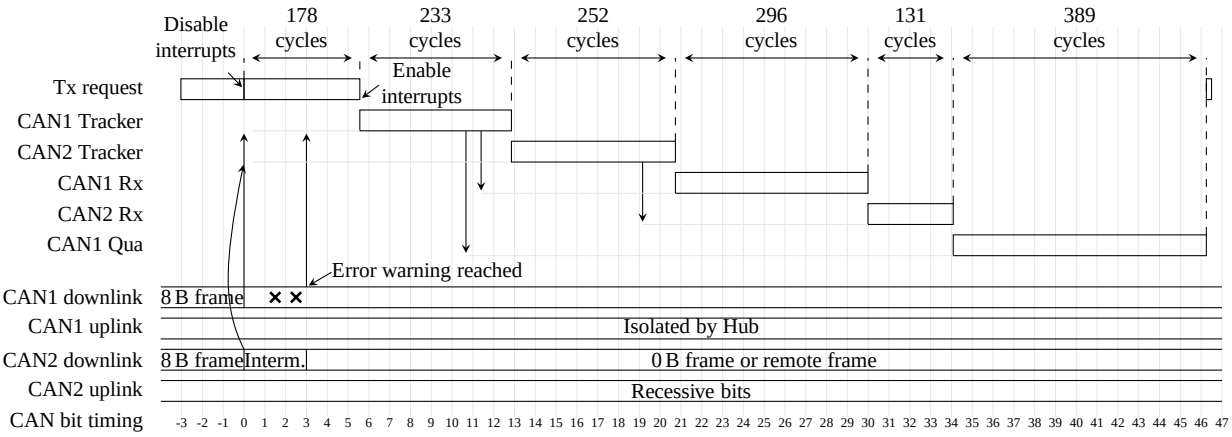

Figure 6. Example scenario that involves the driver's routines

centrate; thus, the driver interface (Section 3) implements only a set of basic initialization and communication primitives, e.g. a primitive to initialize the driver, a primitive to request a transmission, and a primitive to read received data. There is also a primitive to tell the application that communication is no longer possible because all controllers are faulty.

Concerning the management routines, the requirement that the qua routine is executed after a pending rx routine (Section 3) is accomplished by assigning to the qua routine a lower second order priority, which is used to disambiguate which interrupt among several pending interrupts with the same priority is served first. Additionally, we observed that both CAN controllers quasi-simultaneously notify about each delivery event and, thus, we set the tx/rx routines' parameter K (Section 4.1) to 0.

The basis for all our tests were 3 programs, i.e. the *3BitCounter*, the *Blinker*, and the *Receiver*, which ran simultaneously, each one on a different node. The channel utilization was maximized, i.e. the separation between each pair of consecutive frames was the minimum intermission period (3 bits). The bit rate was of 921.25 Kbps, instead of CAN's maximum 1 Mbps [3], due to the oscillators we used for the hubs and nodes. The 3BitCounter executes an infinite loop that increases a 3-bit counter, displays the counter's value on the 3 "least significant" LEDs of the dsPICDEM board, and transmits it in the 3 least significant bits of the payload of a 1-byte-data frame (all other bits of the payload are set to zero). The Blinker executes an infinite loop where it increases a 1-bit counter, displays the counter's value in the *most-significant LED* of its dsPICDEM board, and transmits it in the fourth most significant bit of the payload of a 1-byte-data frame (the rest of the payload is set to zero). The Receiver continuously receives any frame transmitted, and displays on its 4 LEDs the binary value received in the 4 least significant bits of the last frame's payload; thus its 3 least significant LEDs count in binary, whereas its most significant LED blinks.

To inject faults at uplinks and downlinks, we implemented, in VHDL, a fault injector for each hub. With it we can inject permanent stuck-at faults at a precise instant of time during a frame broadcast and during bus-idle.

5.1 Fault-tolerance tests

We tested our ReCANcentrate nodes under scenarios where a single permanent stuck-at-recessive or stuck-at-dominant fault is injected into a single downlink or uplink. Tests involving various bit-flipping patterns, controller crashes, multiple faults, or inconsistency scenarios are postponed for future work.

Although there are four injection points (two downlinks and two uplinks) and two types of faults, there are more than $2 \cdot 4 = 8$ relevant fault injection tests. Note that the role of the affected controller (transmitter/receiver) and the timing of when exactly a permanent fault is injected is relevant. A stuck-at-recessive affecting a downlink will corrupt an on-going frame only if it is injected between the Start Of Frame and the ACK Delimiter of that frame; otherwise, the affected controller will not detect any error and will not globalize it. Similarly, a stuck-at-recessive at the uplink of a controller that is transmitting will not corrupt the on-going frame if it does not affect any dominant bits previous to the ACK delimiter. Moreover, a stuck-at-recessive affecting a downlink/uplink will not be noticed by the affected controller as long as that controller does not attempt a transmission or, when the uplink is stuck-at-recessive and thus it cannot transmit ACKs, as long as it observes that another controller sends the ACK bit.

Taking into account the controller's role and the time of injection, we set up 20 fault-injection tests, each executed at least 10 times, which gave us confidence of the fault-tolerance capabilities of our driver. However, note that these tests do not cover all possible fault scenarios, i.e. all possible combinations of faults and actions carried out to tolerate them. To cover all of them we plan to use model checking techniques [11].

In all our tests the system behaved as expected and faults were tolerated: each fault never noticeably disturbed the applications executed at the nodes, i.e. the 3BitCounter, the Blinker, and the Receiver. These tests are summarized in Tables 1 and 2. For instance, test 7 of Table 1 corre-

Table 1. Stuck-at recessive fault injection tests

	Node	Where	When	Observed consequences and fault-tolerance actions
1	Receiver	tx ctrl downlink	EOF	no global error; tx ctrl not quarantined (sees medium as idle); receptions continue at non-tx ctrl
2	Receiver	non-tx ctrl downlink	EOF	no global error; non-tx ctrl not quarantined (sees medium as idle); receptions continue at tx ctrl
3	Receiver	tx ctrl downlink	data field	temporarily global error; hub isolates tx ctrl uplink; tx ctrl is quarantined; non-tx ctrl becomes new tx ctrl; receptions continue at new tx ctrl
4	Receiver	non-tx ctrl downlink	data field	temporarily global error; hub isolates non-tx ctrl uplink; non-tx ctrl is quarantined; receptions continue at tx ctrl
5 / 6	Blinker	tx ctrl downlink	EOF / data field	temporarily global error; hub isolates tx ctrl uplink; tx ctrl is quarantined; non-tx ctrl becomes new tx ctrl; if a frame was pending in old tx ctrl, qua routine instructs its transmission through the new tx ctrl
7	Blinker	non-tx ctrl downlink	EOF	no global error; non-tx ctrl not quarantined (sees medium as idle); *maxIncon* retransmissions through tx ctrl
8	Blinker	non-tx ctrl downlink	data field	temporarily global error; hub isolates non-tx ctrl uplink; non-tx ctrl is quarantined; transmissions continue at tx ctrl
9	Receiver	tx ctrl uplink	arbitrary	no global error; tx ctrl is not quarantined (no bit error during ACK because tx ctrl receives non-tx ctrl's ACK); both controllers continue to receive
10	Receiver	non-tx ctrl uplink	arbitrary	no global error; non-tx ctrl is not quarantined (no bit error during ACK because non-tx ctrl receives tx ctrl's ACK); both controllers continue to receive
11	Blinker	tx ctrl uplink	arbitrary	no global error; tx ctrl is quarantined; non-tx ctrl becomes new tx ctrl; if a frame was pending in old tx ctrl, qua routine instructs its transmission through the new tx ctrl
12	Blinker	non-tx ctrl uplink	arbitrary	no global error; non-tx ctrl is not quarantined (no bit error during ACK because non-tx ctrl receives ACK from another node); receptions continue at non-tx ctrl

sponds to the injection of a permanent stuck-at-recessive fault in the downlink of the non-tx controller of the Blinker node during the EOF of a frame. This fault does not generate any channel error, but each time the tx controller of this node completes a frame transmission, its driver observes that the non-tx controller omits the reception of that frame. Thus, as long as the non-tx controller remains active, each frame transmission requested by the Blinker is retransmitted *maxIncon* times by its tx controller and, only then, the driver notifies the Blinker about the successful transmission. Another interesting test is the one specified in row 19 of Table 2, in which a permanent stuck-at-dominant is injected in an arbitrary instant of time in the uplink of the tx controller of the Blinker. This always provoked a global error in the channel blocking any communication until the hub isolates the Blinker's tx controller. Then, all other controllers start communicating normally and, at a certain point in time, the driver diagnoses the tx controller of Blinker as faulty. This triggers Blinker's qua routine, which declares the non-tx controller as the new tx controller and, if necessary, requests through the new tx controller the transmission of the frame pending at the old tx controller.

5.2 Performance tests

The driver performance is an issue of main concern since it must handle each delivery event and reset the respective tracking variables before a new delivery event occurs; otherwise, a management routine corresponding to a new delivery event could incorrectly cooperate with routines that are handling a previous one.

On the one hand, note that before a routine consults the tracking variable that indicates if the other routine is also going to be executed, it must wait enough time to allow the other routine to be triggered. We think that this time should be very short, of the order of a fraction of the bit time.

To validate the driver performance in absence of faults, we simultaneously ran the above-mentioned three test programs during sessions of more than 8 hours, maximizing the channel utilization at 921.25 Kbps. We corroborated, by means of the corresponding assertions in the code, that all delivery events where timely managed. Moreover, all delivery events were also punctually processed in all fault injection tests.

Also, we measured the execution time of the driver for the estimated worst-case scenario to check if it can be timely managed. This scenario was introduced before in Section 4.3. It happens when, just after an 8-byte-data CAN frame finishes, the node needs to execute an 8-byte tx req, the CAN 1 tracker, the CAN 2 tracker, the CAN 1 rx, and the CAN 2 rx routine before the shortest CAN frame (starting just after the minimum, 3-bit, intermission period) is broadcast—the qua routine executed last can finish later. The relevant execution times measured in this scenario for the functions that must finish timely (shown in Figure 6) are 178, 233, 252, 296, and 131 instruction cycles respectively, totaling 1090 cycles. The shortest cycle is 33.92 ns, so 32 cycles can be executed at most per bit at 921.25 Kbps. This gives, in the worst case, where the shortest frame (44 bits) follows, the driver $(44 + 3) \cdot 32 = 1504$ instruction cycles to manage the reception, which is greater than the 1090 cycles consumed by the functions that must finish timely.

6 Conclusions and future work

In this paper we present the design, implementation and a first experimental validation of a software driver that executes at each ReCANcentrate node and that allows CAN-

Table 2. Stuck-at dominant fault injection tests

	Node	Where	When	Observed consequences and fault-tolerance actions
13	Receiver	tx ctrl downlink	arbitrary	temporarily global error; hub isolates tx ctrl uplink; tx ctrl is quarantined; non-tx ctrl becomes new tx ctrl; receptions continue at new tx ctrl
14	Receiver	non-tx ctrl downlink	arbitrary	temporarily global error; hub isolates non-tx ctrl uplink; non-tx ctrl is quarantined; receptions continue at tx ctrl
15	Blinker	tx ctrl downlink	arbitrary	temporarily global error; hub isolates tx ctrl uplink; tx ctrl is quarantined; non-tx ctrl becomes new tx ctrl; if a frame was pending in old tx ctrl, qua routine instructs its transmission through the new tx ctrl
16	Blinker	non-tx ctrl downlink	arbitrary	temporarily global error; hub isolates non-tx ctrl uplink; non-tx ctrl is quarantined; transmissions continue at tx ctrl
17	Receiver	tx ctrl uplink	arbitrary	temporarily global error; hub isolates tx ctrl uplink; tx ctrl is not quarantined (no bit error during ACK because tx ctrl receives non-tx ctrl's ACK); both controllers continue to receive
18	Receiver	non-tx ctrl uplink	arbitrary	temporarily global error; hub isolates non-tx ctrl uplink; non-tx ctrl is not quarantined (no bit error during ACK because non-tx ctrl receives tx ctrl's ACK); both controllers continue to receive
19	Blinker	tx ctrl uplink	arbitrary	temporarily global error; hub isolates tx ctrl uplink; tx ctrl is quarantined; non-tx ctrl becomes new tx ctrl; if a frame was pending in old tx ctrl, qua routine instructs its transmission through the new tx ctrl
20	Blinker	non-tx ctrl uplink	arbitrary	temporarily global error; hub isolates non-tx ctrl uplink; non-tx ctrl is not quarantined (no bit error during ACK because non-tx ctrl receives ACK from another node); receptions continue at non-tx ctrl

based applications and protocols to transparently use this star infrastructure. We explain the driver's main management routines, focusing on how they interact. Then, we describe its first implementation on top of a real ReCANcentrate prototype. We experimentally corroborated the driver's correct operation in the absence and in the presence of faults we physically injected. However, we showed that it is impossible to cover all fault scenarios by means of exhaustive fault injection and, thus, we plan to use model checking techniques to overcome this limitation. We also observed that the driver timely performed in all the experiments we carried out. Moreover, we estimated the worst-case scenario in terms of performance and we experimentally characterized its execution time, which indicated that the driver—even though its code is not optimized—can perform timely.

We plan to explore all possible execution scenarios to ascertain a guaranteed upper bound of the worst-case execution time and, then, to prove that the driver can perform at the maximum CAN bit rate; demonstrate, experimentally and through model checking, that the driver can deal not only with stuck-at, but also with bit-flipping faults and CAN inconsistency scenarios; and add features to the driver to reintegrate temporarily faulty CAN controllers.

7 Acknowledgement

This work was supported by the Spanish Science and Innovation Ministry with grant DPI2008-02195, the FCT/CERN grant PTDC/EEA-ACR/73307/2006 and the European Union grant SCP8-GA-2009-233715.

References

[1] G. Bauer, H. Kopetz, and W. Steiner, "The Central Guardian Approach to Enforce Fault Isolation in the Time-Triggered Architecture", in *Proceedings of the The Sixth International Symposium on Autonomous Decentralized Systems (ISADS'03), Washington, DC, USA*, 2003, p. 37. IEEE Computer Society.

[2] FlexRay^{TM}, "FlexRay Communications System - Protocol Specification, Version 2.1", 2005.

[3] ISO, "ISO11898. Road vehicles - Interchange of digital information - Controller Area Network (CAN) for high-speed communication", 1993.

[4] M. Barranco, L. Almeida, and J. Proenza, "ReCANcentrate: A replicated star topology for CAN networks", *ETFA 2005. 10^{th} IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy*, 2005.

[5] M. Barranco, J. Proenza, and L. Almeida, "Reliability Improvement Achievable in CAN-based Systems by Means of the ReCANcentrate Replicated Star Topology", in *To be published in WFCS 2010. 8^{th} IEEE International Workshop on Factory Communication Systems, Nancy, France*, 2010.

[6] M. Barranco, J. Proenza, and L. Almeida, "Designing and Verifying Media Management in ReCANcentrate", in *WFCS'08. IEEE Workshop on Factory Communication Systems, Dresden, Germany*, 2008.

[7] M. Barranco, D. Gessner, J. Proenza, and L. Almeida, "Demonstrating the feasibility of media management in ReCANcentrate", in *ETFA 2009. 14^{th} IEEE International Conference on Emerging Technologies and Factory Automation, Palma de Mallorca, Spain*, 2009.

[8] J. Proenza and J. Miro-Julia, "MajorCAN: A modification to the Controller Area Network to achieve Atomic Broadcast", *IEEE Int. Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.

[9] Microchip Technology Inc., *dsPIC30F6011A / 6012A / 6013A / 6014A Data Sheet - High-Performance, 16-Bit, Digital Signal Controllers*, 2006.

[10] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 2004.

[11] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell", *Int. Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.