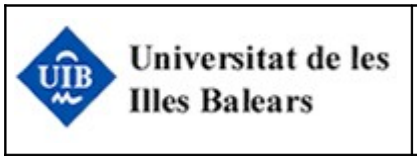

ESCOLA POLITÈCNICA SUPERIOR

ESCOLA POLITÈCNICA SUPERIOR

UNIVERSITAT DE LES ILLES BALEARS

PROJECTE DE FINAL DE CARRERA

Estudi :

Enginyeria Informàtica

Títol:

Construction of a Hardware Prototype of ReCANcentrate and Implementation of a Media Management Driver for the Nodes of the Prototype

Alumne: David Gessner

Directors : Julián Proenza Arenas
Manuel Alejandro Barranco González

Data: Novembre de 2010

Fem constar que el projecte de final de carrera titulat *Construction of a hardware prototype of ReCANcentrate and implementation of a media management driver for the nodes of the prototype* ha estat realitzat, sota la direcció de Julián Proenza Arenas i Manuel Alejandro Barranco González, per David Gessner. Així mateix declaram que el projecte està finalitzat i preparat per la seva presentació pública.

Palma, novembre de 2010

Signat: David Gessner
Projectista

Signat: Julián Proenza Arenas
Co-director del projecte final de carrera

Signat: Manuel Alejandro Barranco González
Co-director del projecte final de carrera

Resum

El protocol *Controller Area Network* (CAN) és amplament utilitzat en sistemes de control distribuïts. Però a pesar del seu ús extens, hi ha una controvèrsia sobre lo adequat que és CAN per sistemes que requereixen un grau de fiabilitat elevat. Aquesta controvèrsia es deguda principalment a una sèrie de limitacions de fiabilitat que té CAN, causades principalment per la topologia en bus que empra. Per superar aquestes limitacions, previ al projecte s'havien dissenyat dues topologies en estrella per CAN. La primera, *CANcentrate*, és una topologia en estrella simple amb un element central denominat concentrador (*hub*) que proporciona mecanismes per millorar la contenció d'errors. La segona, *ReCANcentrate*, és una topologia en estrella replicada, amb dos concentradors, que proporciona tolerància a fallades a més de proporcionar contenció d'errors.

Aquest projecte consisteix en la implementació d'un nou prototip de *ReCANcentrate*. Aquest nou prototip està basat en un prototip anterior que consistia en mostrar la viabilitat de implementar amb components comercials (*commercial off-the-shelf*) els concentradors de ReCANcentrate, així com mostrar que els mecanismes de contenció d'errors i tolerància a fallades dels concentradors funcionen correctament. El nou prototip en canvi té com a objectiu mostrar la viabilitat i la capacitat de tolerància a fallades dels nodes. Concretament, l'objectiu d'aquest projecte és la construcció del hardware d'un nou prototip, que consisteix en dos concentradors i tres nodes; la implementació d'un programa que faci de *driver* per als nodes del prototip; i la verificació, mitjançant experiments, de que els nodes poden tolerar tota una sèrie de fallades introduïdes mitjançant injecció de fallades.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Background and motivation

The Controller Area Network (CAN) protocol was designed for control applications inside automobiles, but nowadays it is used in many other applications as well. For instance, CAN is now used as a fieldbus, that is, to interconnect field devices such as sensors and actuators in a system such as a manufacturing plant; it is used as a communication protocol inside many vehicles which are not automobiles, such as trains, ships, and aircraft; and it is used in many machines, such as medical equipment, household appliances, elevators, and escalators [Voss, 2005]. But despite being widely used, according to Barranco, Proenza, Rodríguez-Navas, and Almeida [2005b], there is a controversy about how suitable CAN is for applications which have very demanding reliability requirements. To completely understand why there is this controversy, one must understand what the reliability limitations of CAN are. We describe these limitations in Chapter 3, Section 3.4.

To overcome CAN's reliability limitations, several researches have worked on improvements of CAN. Among them are the already cited Julián Proenza, Guillermo Rodríguez-Navas, and Manuel Barranco. All three are members of the *Systems, Robotics and Vision* (SRV) group of the *Departament de Matemàtiques i Informàtica* of the *University of the Balearic Islands* (UIB). Their stated goal is to make CAN suitable for applications with very demanding reliability requirements, while taking advantage of its main features: very low cost, good real-time performance, and already good reliability [Proenza, 2009]. So far, together with other researchers, they have already designed, built as a prototype, and tested several improvements [Barranco, Proenza, Rodríguez-Navas, and Almeida, 2006b; Proenza, 2007], including ReCANcentrate [Barranco, Proenza, and Almeida, 2006a], the architecture for which we built the new prototype described in this report.

Before we explain why we built this new prototype if one had already been built, let us briefly describe what ReCANcentrate is. The basic idea behind the ReCANcentrate architecture is to use a *replicated active star topology* instead of CAN's original bus topology [Barranco et al., 2005a]. *Star topology* means that each node is connected to a central element through its own link. This central element is called a *hub* in ReCANcentrate. *Replicated* means that there is not just one hub, which would be a *single point of failure*, but multiple hubs. In Figure 1.1 we can see a depiction of the ReCANcentrate architecture for the case of three nodes and two hubs. Each node is connected to each of the hubs by a separate *link*. Moreover, the hubs themselves are interconnected through *interlinks*, and this is done redundantly. This provides full redundancy with no single points of failure in the communication medium. Finally, *active* means that a hub converts the electrical signals it receives from the nodes and the other hub to logical values (0s and 1s) and couples these values on a logical level before transmitting the result back to the nodes (a passive star, in contrast, merely couples the incoming signals on the electrical level,

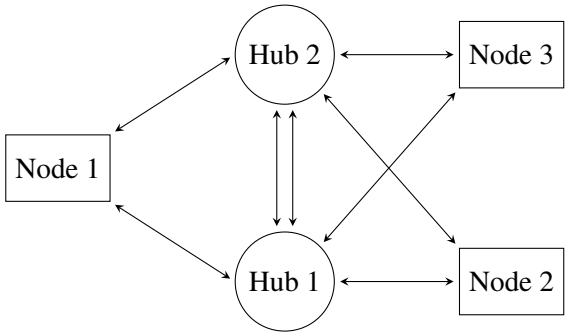

Figure 1.1.: Architecture of ReCANcentrate for three nodes and two hubs. Each node is connected to each of the hubs through a separate *link* and the hubs themselves are interconnected through *interlinks*. The result is the elimination of all single points of failure in the communication medium. (Based on a figure by Barranco et al. [2005a].)

which has some problems such as coupling losses). Chapter 5 describes ReCANcentrate in more detail.

Given this brief overview of ReCANcentrate, we can now point out why we built a new prototype and why its predecessor was not sufficient. The goal of the predecessor was to demonstrate the feasibility of the ReCANcentrate hubs. For this, Barranco et al. used what we call *simplified nodes*. What do we mean by simplified nodes? Since in ReCANcentrate the hubs couple each others traffic and broadcast the result back to the nodes, each node receives the same traffic twice, once from each hub. Moreover, in principle, a node can transmit its messages through either hub. Because of this, it is necessary to provide the nodes with adequate *media management*, that is, the nodes must implement adequate mechanisms to transmit and receive through the replicated star while tolerating faults. This can be achieved using several approaches. The one used in the predecessor prototype is a particularly easy one to implement (the approach is described in Section 5.4.3). Unfortunately, however, this approach does not have all the fault tolerance mechanisms specifically designed for ReCANcentrate. It has therefore some limitations in its media management (which are also described in Section 5.4.3). It is because of these limitations that we call the nodes used by Barranco et al. *simplified nodes*. Despite these limitations, the simplified nodes were adequate for the previous prototype. This is so because the focus was on testing the hubs: the nodes' main function was simply to exercise the hubs while the hubs' fault tolerance was tested. Nevertheless, it remained to be demonstrated that ReCANcentrate also works properly with nodes that implement all of the fault tolerance mechanisms designed for ReCANcentrate and which, contrary to the simplified nodes, take full advantage of the replicated media. The difference then between the previous prototype and the one we have implemented is that our prototype did not use simplified nodes. Moreover, as the hubs had already been thoroughly tested in the previous prototype, in our prototype the focus has been on the testing of the nodes.

As a starting point for the new prototype we had information about the previous one [Barranco et al., 2006a], a design of the architecture of the nodes, and an initial design of several routines to be implemented as the nodes' driver [Barranco, Proenza, and Almeida, 2007, 2008].

1.2. Goal of the project

The goal of our project was to build and test a new prototype of *ReCANcentrate*, a replicated star topology for the *Controller Area Network* (CAN) protocol. More precisely, the goal was to build the hardware for the prototype, which is comprised of two ReCANcentrate hubs and three ReCANcentrate nodes; to write a software driver for the ReCANcentrate nodes; and to thoroughly test the nodes under fault-free conditions as well as when faults are injected. Although our prototype was not the first prototype of ReCANcentrate—as already indicated, it was the successor of a previous one by Barranco et al. [2006a]—it was the first prototype to implement nodes which have all the fault tolerance mechanisms designed for ReCANcentrate.

1.3. Tasks realized

This section outlines the tasks realized in our project. Together with the next section, this should give a clear overview of which tasks should be attributed to this project and which should not.

Study of relevant documentation

To build our prototype we had to understand how ReCANcentrate works. This requires a series of concepts from the fault tolerance field and a clear understanding of CAN. Moreover, it is very useful to comprehend ReCANcentrate's precursor, CANcentrate, before studying ReCANcentrate itself. As a result, there was a non-negligible amount of literature which had to be studied before starting with the implementation of the prototype. This literature is referenced throughout this report and is listed in the bibliography at the end of this document.

Viability study

To implement the prototype we had to choose some hardware components from which to built it. For the hubs and their input and output (I/O) modules, which are used to connect the hubs to the nodes and to each other, the choice was clear. We used the same components that were used in the previous prototype [Barranco et al., 2006a]. We knew these components were adequate for the hubs and, moreover, they allowed us to reuse the VHDL description that had already been written for the hubs (VHDL is a hardware description language used to configure FPGAs, see Section 5.4.2 for more details). For the nodes, on the other hand, the choice was not so clear. Each node is basically composed of a microcontroller and an I/O module used to connect each node to the two hubs. The I/O module could remain the same as in the previous prototype, but the microcontroller used for the nodes had to satisfy different requirements. Therefore we had to choose a new microcontroller for the nodes (we chose the dsPIC30F6014A from Microchip) and carry out a viability study, which can be found in Section 7.3, to assess if this new microcontroller would satisfy the new requirements.

Building the hardware prototype

Once we had chosen the hardware components to use, the next task was building the prototype. For this, we used the *wirewrap prototyping technique*. This technique is described in Chapter 7 along with the task of building the prototype.

Enabling a third port on the ReCANcentrate hubs

Manuel Barranco provided the VHDL source code that was used in the previous prototype for the ReCANcentrate hubs. This code only implemented ports for two nodes, but the new prototype had three nodes; therefore an additional port had to be implemented in the code.

Helping to improve the design of the driver

For the nodes to correctly manage the replicated media, they had to execute a software driver specifically designed for this purpose. Barranco et al. [2007, 2008] have published a few papers that sketch out the main routines of such a driver. Moreover, Manuel Barranco provided an initial design of these routines in form of several flowcharts (shown in Appendix A), and our job was to take them as the specification for a driver customized for the nodes' microcontrollers. Nevertheless, when we studied these flowcharts, we noticed a few things which could be improved. Thus, together with Manuel Barranco, we improved the initial design. The improved design is described in Chapter 6.

Getting to know the development environment

Each hub of our prototype was implemented on a Field-Programmable Gate Array (FPGA), and the nodes were implemented with dsPIC30F6014A microcontrollers from Microchip. We had to familiarize ourselves with the development environments used to program these two kinds of devices.

Implementation of the driver

Writing the driver for the ReCANcentrate nodes consisted in taking the improved design of the routines and implementing the routines for the dsPIC30F6014A microcontroller. In addition, we also implemented support code to abstract away the hardware details of the microcontroller. This task is described in Chapter 8.

Implementation of a simple API for the driver

In order to write programs for the ReCANcentrate nodes of our prototype, we needed some way to interface with the driver. We therefore wrote a simple application programming interface (API) that allows a user to initialize the nodes and to send and receive CAN messages. We kept the API very simple as it would only be used by us to write test programs and was not intended to be used by a larger programming audience. The API is described in Section 8.4.

Adding a fault-injection module to the ReCANcentrate hubs

In order to test the fault tolerance of the nodes we had to inject faults into the prototype. To accomplish this, the easiest solution was to add a fault-injection module to the hubs. This required no hardware changes because the fault-injection module could be completely implemented in VHDL The specification and implementation of the fault-injection module is described in Section 9.1.1.

Testing the driver

As the development of the prototype progressed we carried out successive tests. The first tests focused on assessing whether the prototype's hardware had been built correctly or not. Later tests, which focused on the driver for the nodes, were executed on a dsPIC30F6014A microcontroller simulator. Finally, the last tests were executed on the physical ReCANcentrate prototype.

In these final tests the nodes were using the driver and the FPGAs of the hubs were loaded with the ReCANcentrate hub configuration, which was generated from a slightly modified version of the VHDL code that Manuel Barranco provided. These last tests have been carried out both under fault-free conditions as well as when faults were injected. More information about the tests is given in Section 7.4 and in Chapter 9.

1.4. Tasks *not* realized

In this section we list a number of tasks that are related to the project but which should not erroneously be attributed to it. Note that this list might not be complete.

- The design of the CANcentrate architecture.
- The design of the ReCANcentrate architecture.
- The implementation in VHDL of the coupler and fault-treatment modules of a ReCANcentrate hub.
- The design of the electronic circuits of Section 5.4.4, which are used to attach external oscillators to the hubs' FPGAs and to implement the links and interlinks.
- The testing of the coupler and fault-treatment modules of a ReCANcentrate hub with "simplified" nodes.
- The formal verification of the driver's routines by means of a model checker.
- The initial design of the driver's routines.

1.5. Overview of the remaining chapters

The remaining chapters have been divided into two parts. Part I gives the necessary foundations needed in order to understand the tasks carried out in this project and describes previous work on which the project is based. Part II describes the tasks which are specific to the project, that is, it describes the tasks we carried out.

Part I contains the following chapters. First, Chapter 2 introduces reliability, fault tolerance, and other related concepts that are relevant to this project. Chapter 3 describes the Controller Area Network (CAN) protocol and its limitations. Chapter 4 presents CANcentrate, explains its fault model and discusses the architecture of the hub and the nodes of a CANcentrate network. Chapter 5 describes ReCANcentrate: it introduces the fault model used for ReCANcentrate, describes the architecture of the hubs and the nodes of a ReCANcentrate network, details the media management strategy used by the nodes (in the absence and presence of faults), describes the architecture and hardware requirements of the node media management driver, and summarizes the previous ReCANcentrate prototype. Moreover, when the previous prototype is described, the chapter also gives a very brief introduction to Field Programmable Gate Arrays (FPGAs).

Part II begins with Chapter 6, which describes the final design of the media management routines of the driver and also includes a few example executions of the media management routines. Part II continues with Chapter 7, which describes the wirewrap prototyping technique, the construction of the new ReCANcentrate hardware prototype, and a series of preliminary tests to assess the correct functioning of the hardware. Chapter 8 contains an explanation of the driver implementation: it briefly introduces the toolchain used during development, details the development methodology used, and summarizes the abstract data types used to implement the driver. Chapter 9 is a description of the tests we executed on the prototype's hardware to determine whether the driver had been implemented correctly and to determine whether the nodes running the driver are capable of tolerating faults. Finally, Chapter 10 gives the conclusions we have reached after finishing the project and includes a personal opinion about the project.

At the end of this report, from page 141 onwards, a series of appendices to this report can be found. They contain the initial design of the media management routines of the driver, the source code files that implement the driver, and the source code files that we used for our test scenarios.

Finally, the bibliography referenced throughout this report and an index can be found on the last few pages of this report.

Part I.

Foundations and previous work

2. Introduction to reliability, fault tolerance, and related concepts

In order to understand the work presented in this report, a basic knowledge of reliability, fault tolerance, and related concepts is necessary. The aim of this chapter is to give this knowledge. Unless otherwise stated, the terminology we adopted is the one presented by Avižienis, Laprie, Randell, and Landwehr [2004].

2.1. Basics

Computing and communication systems—simply referred to as systems from here onwards—have what is called a *function*, which Avižienis et al. define as "what the system is intended to do" [Avižienis et al., 2004].

What a system does to implement its function can be described by a sequence of states. Of these states the part that is perceivable from outside the system is called the system's *external state*. The remaining part is called the system's *internal state*. The union of external state and internal state is the system's *total state*. The *service* delivered by a system is a sequence of its external states.

A system delivers a *correct service* when it implements its function, that is, when the system actually does what it is intended to do. A *service failure*, or simply *failure*, "occurs when the delivered service deviates from correct service" [Avižienis et al., 2004], that is, when the system stops doing what it is intended to do. If a system has more than one function and only a subset of these functions suffers a failure, the system is said to be in a *degraded mode*; moreover, it is said that the system has suffered a *partial failure*.

The *service failure modes*, or simply *failure modes*, are the different ways in which a failure can manifest; examples are the incorrect computation of results, delivering a result too late, giving different results to different *user systems* although they expect the same result (*user systems* are systems that receive a service from another system), and a *crash* failure, which occurs when a system permanently stops delivering its service.

An *error* is "the part of a system's total state that may lead to a failure". "An error is *detected* if its presence is indicated by an error message or error signal"; whereas an error is *latent* if it is "present but not detected". A *fault* is "the adjudged or hypothesized cause of an error" [Avižienis et al., 2004]. Note that there exists a causal relationship between faults, errors, and failures: a fault may generate an error, and an error may generate a failure. If a fault actually does generate an error, then it is *active*; otherwise, it is *dormant* (dormant is also referred to as *passive* by many other authors).

There are many criteria that can be used to classify faults. One criteria we want to highlight is persistence: faults can either be *permanent faults*, their "presence is assumed to be continuous

in time"; or *transient faults*, their "presence is bounded in time" [Avižienis et al., 2004].

A system is comprised of other systems put together to interact; each of these other systems is called a *component*. Errors can propagate within a component, and they can propagate from one component to another. When an error is propagated to the point where the system's service is delivered, a failure of the system occurs.

A system may have some critical components where a fault may directly lead to an overall failure of the whole system—without having to propagate to several other components first. These components are commonly known as *single points of failure*.

Avižienis et al. give several examples illustrating some of the concepts so far introduced in this chapter. Perhaps the following one will help to clarify these concepts:

> A short circuit occurring in an integrated circuit is a *failure* (with respect to the function of the circuit); the consequence (connection stuck at a Boolean value, modification of the circuit function, etc.) is a *fault* that will remain dormant as long as it is not activated. Upon activation (invoking the faulty component and uncovering the fault by an appropriate input pattern), the fault becomes *active* and produces an *error*, which is likely to propagate and create other errors. If and when the propagated error(s) affect(s) the delivered service (in information content and/or in the timing of delivery), a *failure* occurs.

2.2. Reliability and dependability

Reliability is defined as "continuity of correct service". Reliability is therefore a property of a system related to its correct service. Apart from reliability, there are other properties of a system related to its correct service. These other properties are usually taken together in the broader concept of *dependability*—which includes reliability.

Avižienis et al. give two definitions for dependability: an original one and an alternate. The original definition states that *dependability* is "the ability to deliver service that can justifiably be trusted". We argue that this definition is circular because trust is defined as "accepted dependence" and dependence of a system A on a system B is defined as "the extent to which system A's dependability is (or would be) affected by that of System B"; that is, dependability is defined in terms of trust, trust is defined in terms of dependence, and dependence is defined in terms of dependability, the original concept for which we sought a definition. Therefore we prefer the alternate definition: *dependability* is "the ability of a system to avoid service failures that are more frequent or more severe than is acceptable".

The properties of a system related to its correct service that we referred to at the beginning of this section are known as *attributes* of dependability. These attributes are the following:

Availability: "readiness for correct service".

Reliability: "continuity of correct service".

Safety: "absence of catastrophic consequences on the user(s) and the environment".

Integrity: "absence of improper system alterations".

Maintainability: "ability to undergo modifications and repairs".

Note that the relationship between reliability and dependability is that reliability is an attribute of dependability.

Apart from attributes, there are two other notions related to dependability. The first is *threats* to dependability, which were introduced in the previous section, and which are faults, errors, and failures. The second is the notion of *means* to dependability, which are the following:

Fault prevention: defined as "means to prevent the occurrence or introduction of faults". It refers to the attempt of avoiding the introduction of faults in the first place and therefore applies to the development phase of a system.

Fault tolerance: defined as "means to avoid service failures in the presence of faults". This is the mean on which we focus in this report. It is treated in more detail in the next section.

Fault removal: defined as "means to reduce the number and severity of faults". It applies to the development phase and to maintenance actions, and it consists in uncovering faults and removing them.

Fault forecasting: defined as "means to estimate the present number, the future incidence, and the likely consequences of faults".

Figure 2.1 shows a summary of the fundamental concepts related to dependability and their relationships (it is based on Avižienis et al.'s *dependability and security tree*, with the difference that it does not show the attributes that are related exclusively to security).

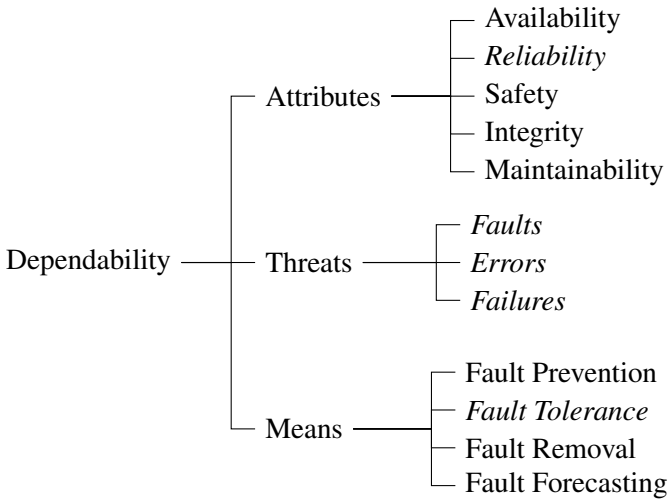

Figure 2.1.: The dependability tree. It summarizes the relationships between the fundamental concepts related to dependability. The concepts shown in *italics* are the ones our project, and ReCANcentrate in general, is mostly concerned with. (The figure is based on the *dependability and security tree* by Avižienis et al. [2004].)

2.3. Fault tolerance

As mentioned in the previous section, the goal of fault tolerance is to avoid failures even when faults are present. This is achieved through *error detection* and *system recovery* (the latter is also referred to as *error recovery* by some authors). Figure 2.2 gives an overview of the tasks involved in fault tolerance.

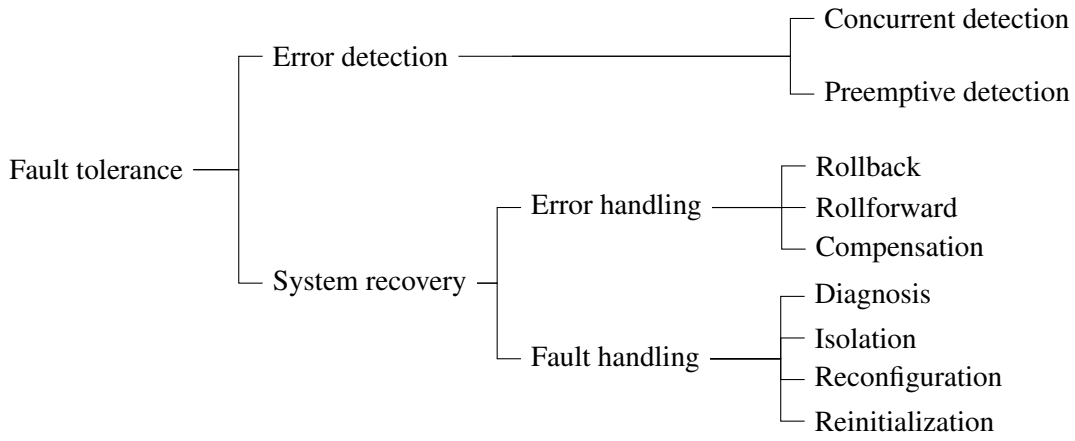

Figure 2.2.: Fault tolerance techniques.

Error detection is the identification of an error's presence. It can be divided into two classes:

Concurrent error detection, which "takes place during normal service delivery".

Preemptive error detection, which "takes place while normal service delivery is suspended" and which "checks the system for latent errors and dormant faults".

System recovery is the transformation of "a system state that contains one or more errors and (possibly) faults into a state without detected errors and without faults that can be activated again" [Avižienis et al., 2004]. System recovery consists of error handling and fault handling:

Error handling: "eliminates errors from the system state" [Avižienis et al., 2004]. It is also referred to as *error processing* by other authors. There are three possibilities to carry out error handling:

Rollback: consists in bringing the system back to a previously saved state (rollback is also known by other authors as *backward recovery*.)

Rollforward: consists in replacing the erroneous state not with a previously saved state but with a new state (rollforward is also known by other authors as *forward recovery*.)

Compensation: consists in having enough redundancy in the erroneous state to allow error masking, that is, to conceal the error so that a correct service can still be provided. Note that as more errors are compensated less redundancy remains available—this is known as *redundancy attrition* [Proenza, 2007].

Fault handling: "prevents faults from being activated again" [Avižienis et al., 2004]. (Fault handling is also referred to as *fault treatment* by other authors.) Fault handling involves four steps:

Diagnosis: consists in the identification and recording of the location and type of faults that have caused the errors from which we want to recover.

Isolation: consists in the "physical or logical exclusion of the faulty components" so that they no longer participate in service delivery, that is, it makes the identified faults dormant [Avižienis et al., 2004]. (Authors that use the term passive for dormant also often refer to fault isolation as *fault passivation*.) Isolation provides *error containment*: it prevents errors from propagating from faulty components to other components [Johnson, 1989].

Reconfiguration: consists in discarding faulty components and using spare components instead, or, if no spare components are available, it consists in reassigning the tasks of the faulty components to non-faulty components.

Reinitialization: consists in checking, updating, and recording the new configuration that resulted from a reconfiguration and in updating system tables and records.

2.4. Implementation of fault-tolerant systems

When designing a fault-tolerant system we have to make assumptions about the different failure modes of the system's components. These assumptions are often referred to as the *fault model* of a system [Johnson, 1989]. The fault model may also consider the duration of faults: we may assume that faults are permanent or that they are transient. A *permanent fault* "remains in existence indefinitely if no corrective action is taken"; whereas a *transient fault* "can appear and disappear within a very short period of time" [Johnson, 1989]. Given a fault model, the fault-tolerance mechanisms of a system are designed so that they are capable of handling the faults considered in the system's fault model.

It is important that the assumptions made in the fault model are realistic, that is, that the system's real failure modes are not too different from the assumed failure modes; otherwise, the system may fail in ways that the implemented fault-tolerance mechanisms are not prepared to handle, resulting in a failure of the whole system. This leads us to the notion of *fault assumption coverage*: a measure of how much the fault assumptions differ from the faults that really occur. A higher fault assumption coverage means that the difference between the assumed faults and the real faults is smaller; whereas a smaller fault assumption coverage means that the difference between real and assumed faults is bigger.

To maximize the fault assumption coverage we could include in our fault model all possible failure modes and types of faults. This, however, would lead to a complex and costly system. It is therefore generally more practical to restrict the failure modes and types of faults included in the fault model. Nevertheless, the fault model should not be restricted too much because that would lead to a fault assumption coverage that is too low to be acceptable, that is, the fault-tolerance mechanisms implemented based on the fault model would not handle most of the faults that do occur [Proenza, 2007].

Fortunately, when designing a fault-tolerant system, we can not only increase the fault assumption coverage by making the fault model broader, but we can also increase the fault assumption coverage by restricting the failure modes of the system. This is done through mechanisms introduced when designing the system that force faulty components to fail in a specific way [Proenza, 2007]. For instance, a specific fault-tolerance mechanism may force a component to not output any result when the result would be wrong; in that case our fault model would not have to include incorrect computations by that component, and the fact that it is not included would not reduce the fault assumption coverage.

3. Controller Area Network (CAN)

Controller Area Network, more commonly known by its acronym CAN, is a serial bus originally developed by the *Robert Bosch GmbH* in the 1980s for automotive applications.

From the 1970s onwards the automotive industry began to include more and more electronics in their automobiles, such as "ABS braking, active suspension, electronic transmissions, automated lighting, air-conditioning, security, and central locking" [Catsoulis, 2005]. These systems often need to exchange information and therefore some means for their interconnection had to be provided. It quickly became very clear that direct point-to-point connections between these systems would no longer be a viable solution: the more components are added, the more direct point-to-point connections are needed, leading to a fast increase in complexity and amount of cabling needed. This increases the cost and weight of the automobile while decreasing its reliability and ease of maintenance. Therefore, some other solution had to be found, and the obvious solution was a low-cost digital network [Catsoulis, 2005].

The particular network that was widely adopted by the automotive industry is CAN, which, as we said, had been specifically designed to satisfy the requirements of automotive applications. These requirements are low cost, real-time response, and the need to function in the electromagnetically harsh and noisy environment found inside an automobile. Because these requirements can not only be found in automotive applications, CAN has spread to many other applications where electromagnetic robustness, real-time, or simply a very inexpensive, not too data intensive, communication system is needed. These applications include factory, home, and building automation; in-vehicle communication in almost any type of vehicle; and the communication inside almost any type of machinery. This widespread use has caused CAN's price to drop to such a low level that, wherever it is applicable, no other protocol can compete.

For CAN only the last two layers of the ISO/OSI reference model are specified (see Figure 3.1). The *Data Link Layer* was specified in 1991 when Bosch published the CAN 2.0 specification [Bosch GmbH, 1991]; whereas the *Physical Layer* was not specified until later, in 1993, when ISO standardized both the Physical Layer and the Data Link Layer in the ISO 11898 standard. Nowadays, both layers are commonly implemented in silicon, either as a stand-alone CAN controller or as an integrated part of a microcontroller [Voss, 2005]—although the Physical Layer is also often implemented in a separate transceiver.

In this chapter we summarize those aspects of the CAN Physical and Data Link Layers which are more relevant to our project. Details not relevant to the project are omitted.

3.1. CAN Physical Layer

There are several specifications for the CAN Physical Layer, but we will only overview the CAN Physical Layer according to the ISO 11898-2 specification, which is the one implemented by the

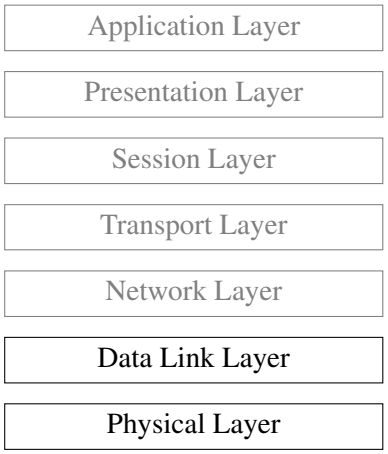

Figure 3.1.: The ISO/OSI seven layer reference model. For CAN only the Data Link Layer and the Physical Layer are specified.

PCA82C250 CAN transceivers [Philips Semiconductors, 1997] that we used in our prototype. (We used these transceivers because they had already been used successfully in the previous prototype by Barranco et al. [2006a].)

In compliance with the ISO 11898-2 specification, the physical CAN bus uses a differential voltage between two wires, known as *CANH* and *CANL*. That is, the voltage on the CAN bus that determines the value on the bus is

$$V_{\text{diff}} = V_{\text{CANH}} - V_{\text{CANL}}$$

where V_{CANH} is the voltage on the CANH wire and V_{CANL} is the voltage on the CANL wire. As illustrated in Figure 3.2, there are two possible bus levels: either $V_{\text{CANH}} = V_{\text{CANL}} = 2.5$ Volts, and consequently $V_{\text{diff}} = 0\,\text{V}$; or $V_{\text{CANH}} = 3.5\,\text{V}$ and $V_{\text{CANL}} = 1.5\,\text{V}$, and consequently $V_{\text{diff}} = 2\,\text{V}$ [Voss, 2005].

The main advantage of a differential voltage for the bus level is that it provides very good resistance to electromagnetic interferences (EMI): if an EMI affects the CAN bus, then both of its wires will be affected almost equally, and, as a consequence, the value of the bus level V_{diff} will remain almost unchanged [Voss, 2005]. An example of this behavior is shown in Figure 3.3.

If the CAN controller used to connect a microcontroller to a CAN bus does not implement the Physical Layer (as it is the case of the CAN controllers we used in our prototype), then a separate CAN transceiver is needed. During reception, the CAN transceiver adapts the differential voltage from the bus to logic level signals that the CAN controller can understand. Similarly, during a transmission, the CAN transceiver adapts the logic level signals from the CAN controller to the differential voltage expected on the bus. Figure 3.4 shows how a CAN controller is connected to a CAN bus by using a transceiver (Txrx) in-between. The figure also shows two resistors of 120 Ohm at each end of the bus; these are known as *terminating resistors* and are used in order to prevent signal reflections.

One of the most characteristic features of CAN is that it physically implements what is known as a *wired-AND* function. As indicated above, the bus can take one of two differential voltage

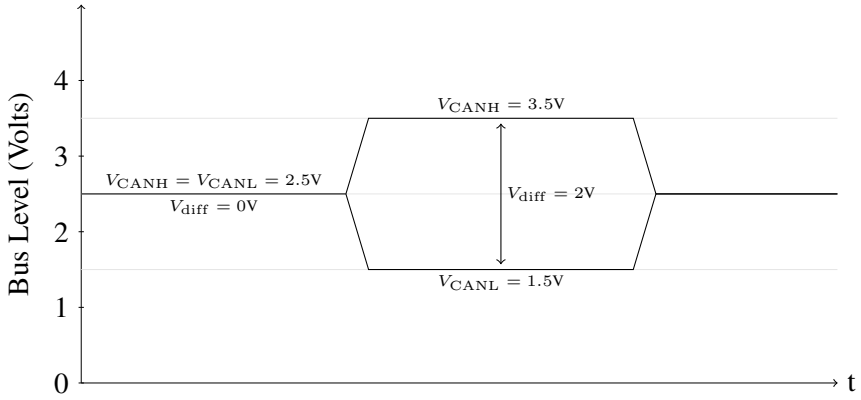

Figure 3.2.: CAN bus level. (Based on a figure by Voss [2005].)

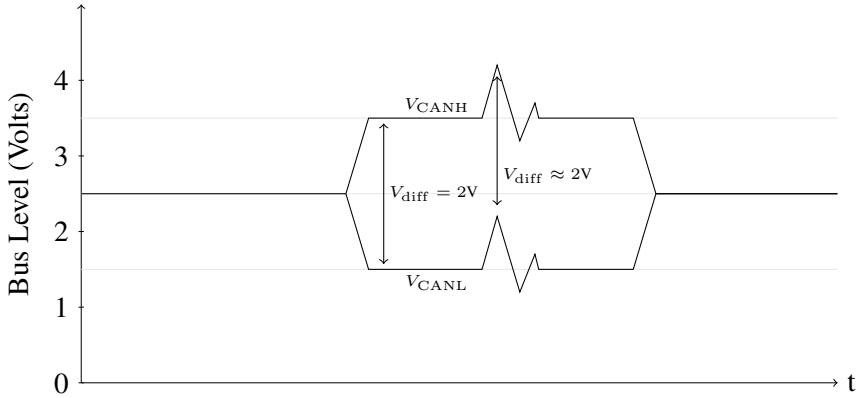

Figure 3.3.: CAN bus level with EMI. The figure shows how an electromagnetic interference (EMI) is likely to affect both the CANH and the CANL wire almost equally. As a consequence the differential voltage between the two wires remains almost unchanged.

levels: $V_{\mathrm{diff}} = 2\,\mathrm{V}$ or $V_{\mathrm{diff}} = 0\,\mathrm{V}$. The first is known as a *dominant* bit value and the second is known as a *recessive* bit value. That is, when a node transmits a dominant bit it creates a voltage difference across the CANH and CANL wires. On the other hand, when a node transmits a recessive bit it does not apply a voltage difference between the CANH and CANL wires. Thus, in the absence of faults, the only way to have a recessive bit value on the bus is when no node applies a voltage difference between the CANH and CANL wires. However, again, in the absence of faults, whenever at least one node applies a voltage difference between the wires, then there will be a dominant bit value on the bus. The effect on the bus is therefore that dominant bits override recessive bits. An example of this behaviour for three nodes is shown in Table 3.1, which, for all possible combinations of dominant ('d') and recessive ('r') bits contributed by the nodes, displays the resulting value on the bus. If we now think of a dominant bit as a logical 0 and of a recessive bit as a logical 1, then we will recognize this table as a truth table of the logical

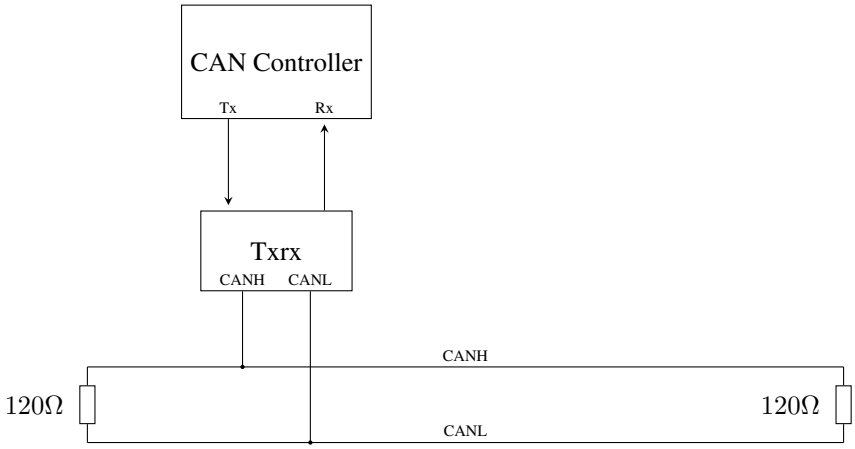

Figure 3.4.: Connecting a CAN controller to a CAN bus through a CAN transceiver (Txrx).

AND operation, where the bit values contributed by the nodes are the terms and the value on the bus is the result of the operation. Because of this, it is said that CAN implements a wired-AND function.

3.2. CAN Data Link Layer

The CAN Data Link Layer can be divided into two sublayers: an Object Layer and a Transfer Layer. Together they provide all the functions of the Data Link Layer defined by the ISO/OSI reference model [Bosch GmbH, 1991].

The Object Layer includes such functions as determining which enqueued message is to be transmitted, which messages received by the Transfer Layer are to be used, and the provision of an interface for the Application Layer. These functions are not detailed in the CAN specification [Bosch GmbH, 1991] and depend on the particular CAN controller.

The Transfer Layer, on the other hand, is completely described in the CAN specification and is the one we overview in this section. It describes the frame format, message arbitration, frame encoding, error signaling, error containment, and bit timing used in CAN.

3.2.1. Frame format

The CAN specification defines four types of frames: *data frames*, *remote frames*, *error frames*, and *overload frames*. This section covers data frames and remote frames. Error frames and overload frames are covered later in this chapter.

Data frames are used to transmit data; remote frames are used to request data. A node sends a data frame when it has data to communicate or when it has been prompted, via a remote frame, to send data. The two frame types are shown in figures 3.5 and 3.6. They can be distinguished from one another by the *Remote Transmission Request* (RTR) bit: in data frames this bit is dominant and in remote frames it is recessive. Besides, data frames have a data field whereas

Contributions			
Node 1	Node 2	Node 3	Value on bus
d	d	d	d
d	d	r	d
d	r	d	d
d	r	r	d
r	d	d	d
r	d	r	d
r	r	d	d
r	r	r	r

Table 3.1.: Resulting value on a CAN bus with three nodes for all possible combinations of dominant ('d') and recessive ('r') bits contributed by the nodes.

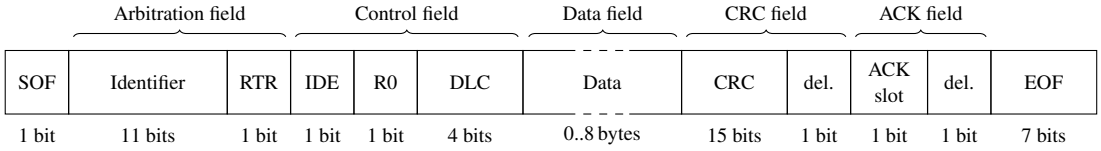

Figure 3.5.: CAN standard *data frame format*. From left to right, a standard data frame comprises the following: a dominant start of frame bit; an arbitration field, which includes a message identifier and a dominant remote transmission request bit; a control field, which includes a dominant identifier extension bit, a dominant reserved bit, and the data length code; a data field; a CRC field, which includes a cyclic redundancy code and a recessive delimiter bit; an acknowledgement field, which includes an acknowledgement slot and recessive acknowledgement delimiter bit; and an end of frame field.

remote frames do not. The remaining fields are identical.

Data and remote frames both begin with a *Start Of Frame* (SOF) bit: a dominant bit that marks the beginning of a new frame. It can easily be recognized on the bus as it contrasts with the recessive bits of an idle bus and with the recessive bits found at the end of each frame (all frames finish with recessive bits). Nodes listening on the bus will synchronize themselves with the node that transmits an SOF. If more than one node transmits an SOF within the same bit time—the amount of time it takes to transmit one bit—then a contention takes place. To resolve this contention, and to decide which node will become the next transmitter, the arbitration field is used.

The *arbitration field* includes the *identifier* and the previously mentioned RTR bit. CAN's *bit-wise arbitration mechanism*, explained in Section 3.2.2, makes sure that after the arbitration field only one node remains as the transmitter. Contrary to many other protocols, the identifier is not the address of a node; instead it is a value that identifies the data transmitted in the message. Receiving nodes can, based on the identifier, decide whether to forward or not the frame to the

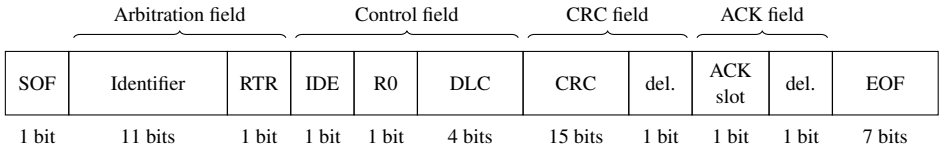

Figure 3.6.: CAN standard *remote frame format*. From left to right, a standard remote frame comprises the following: a dominant start of frame bit; an arbitration field, which includes a message identifier and a recessive remote transmission request bit; a control field, which includes a dominant identifier extension bit, a dominant reserved bit, and the data length code; a CRC field, which includes a cyclic redundancy code and a recessive CRC delimiter bit; an acknowledgement field, which includes an acknowledgement slot and a recessive acknowledgement delimiter bit; and an end of frame field.

application. For instance, in a given CAN network a particular identifier might identify frames that carry data of a particular sensor; if a receiving node is interested in that sensor's data, it will forward the received frame to the application; otherwise, it will discard the frame.

The arbitration field is followed by the *control field*. It includes the *IDentifier Extension* (IDE) bit, a dominant reserved bit called *R0*, and the *Data Length Code* (DLC).

The IDE bit is used to distinguish between frames with *standard frame format* and frames with *extended frame format* (the frames in figures 3.5 and 3.6 are both in standard frame format). The two formats differ in the length of their identifiers: standard frames use 11-bit identifiers, whereas extended frames use 29-bit identifiers. In standard frames the IDE bit is dominant, indicating that no further identifier bits follow; in extended frames the IDE bit is recessive, indicating that further identifier bits do follow. There are further differences, but we will not describe extended frames any further as our project only deals with standard frames[1].

In data frames the DLC indicates the number of bytes included in the data field; whereas in remote frames it indicates the number of data bytes requested. The DLC has 4 bits, nevertheless, its value ranges only from 0 to 8—not up to $2^4 = 16$, as one might expect. The length of the data field is therefore at most 8 bytes.

In data frames the control field is followed by the data field, which in turn is followed by the *CRC field*. In remote frames the control field is directly followed by the CRC field, with no data field in between. The CRC field is composed of a 15-bit *Cyclic Redundancy Code* (CRC) and a single recessive bit called *CRC delimiter*. The former is used for error detection and the latter delimits the CRC field from the next field.

The next field is the *ACK field*, which is composed of an *ACK slot* and an *ACK delimiter*. The ACK slot is a single bit that is transmitted with a recessive value by the transmitter. Receiving nodes that get a correct result when checking the CRC override the transmitter's recessive ACK slot with a dominant bit, thereby indicating to the transmitter that at least one node received

[1] As described in Section 5.3.4, the driver for the ReCANcentrate nodes must have processed the reception or transmission of a frame before the next frame is received. Shorter frames give less time for this than longer frames, thus, the shorter standard frames are the worst-case frame format. To demonstrate the feasibility of ReCANcentrate, the goal of our project, it is therefore enough to demonstrate that our prototype works with standard frames.

the frame without CRC errors. The other component of the ACK field, the ACK delimiter, is a recessive bit that, together with the CRC delimiter, surrounds the ACK slot with recessive values.

Finally, data and remote frames are terminated by an *End Of Frame* (EOF), which consists of 7 recessive bits.

All data frames and remote frames are separated from preceding frames by an *interframe space*—error frames and overload frames (see sections 3.2.4 and 3.2.6), on the other hand, are not preceded by an interframe space. The interframe space starts with 3 recessive bits, known as *intermission*, and continues with the recessive bits of an idle bus. It is therefore of variable length, with the length depending on how long the bus remains idle and depending on whether an overload is signaled or not (overload signaling is covered in Section 3.2.6).

3.2.2. Bit-wise arbitration mechanism

As mentioned in the previous section, when more than one node starts the transmission of a frame in the same bit time, a contention takes place. To resolve this contention and decide which node is allowed to send its frame, CAN's bit-wise arbitration mechanism is used.

This mechanism works as follows. Each transmitting node monitors the actual bits on the bus while it transmits its arbitration field. As soon as a transmitting node detects a dominant bit on the bus while it is transmitting a recessive bit, the node looses the arbitration. A node that has lost the arbitration then backs off, becomes a receiving node, and retries its transmission when the bus becomes idle again. Because in CAN all nodes must use different identifiers, the result is that after the arbitration field only a single transmitter is left; all other nodes will have become receiving nodes. Note that because a recessive bit corresponds to a logical 1 and a dominant bit corresponds to a logical 0, the node that wins the arbitration is the one that transmitted the identifier with the lowest value. Moreover, note that since a remote frame has a recessive RTR bit and a data frame has a dominant RTR bit, data frames have a higher priority than remote frames.

Figure 3.7 shows an example of how the arbitration mechanism works. Nodes 1, 2, and 3 have each a frame to transmit in the same bit-time, typically because they were all waiting for the bus to be idle in order to transmit. Therefore, all three issue a start of frame nearly simultaneously. Node 4 has no frame to transmit and therefore remains in receiving mode, which implies that its contribution to the bus are recessive bits. At each bit-time the value seen on the bus is the logical-AND of the nodes' bits at that bit-time. After having issued a start of frame (SOF) each transmitting node sends its identifier until it detects that, despite having sent a recessive value to the bus, it gets a dominant value from it. From then on the node becomes a receiving node until the bus is idle again. This happens to Node 1 at bit-time 4 and to Node 3 at bit-time 7. After the arbitration field Node 2 is the only transmitter and, although not shown in the figure, it transmits the remaining fields of its frame.

3.2.3. Frame encoding

CAN uses a *Non Return to Zero* (NRZ) bit coding. That means that the value of a bit is determined by the voltage level on the bus, which is kept constant for the duration of the bit, instead

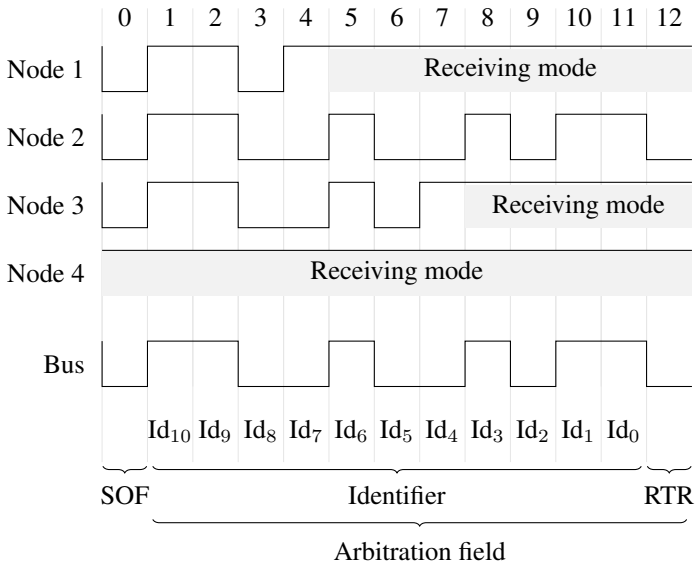

Figure 3.7.: CAN arbitration example.

of being determined by voltage changes. As a consequence, a sequence of equal bits does not cause a voltage change on the bus. Voltage changes, however, are used by receiving nodes to synchronize themselves with the leading transmitter—the node that has won the arbitration and is transmitting its frame. So, to ensure correct synchronization, CAN additionally employs *bit stuffing*: for every sequence of five consecutive bits with the same value, an additional bit, called *stuff bit*, is inserted. In data or remote frames bit stuffing is applied to all the bits from the start of frame up to—and including—the CRC sequence. It is not applied to the CRC delimiter, the ACK field, the end of frame, or the intermission. Also, bit stuffing is not applied to error frames and overload frames.

3.2.4. Error-signaling mechanism

According to the CAN specification [Bosch GmbH, 1991], there are five different types of errors that can be detected by the CAN nodes:

Bit errors: These are errors where a transmitting node detects that the bit on the bus differs from the bit it has sent. Note, however, that there are two exceptions: during the arbitration phase it is not considered a bit error when a dominant bit is detected on the bus while a recessive bit is transmitted, and it is neither considered a bit error when a recessive ACK slot is overridden by a dominant bit.

Stuff errors: These are errors where the bit stuffing is violated; that is, whenever a sixth consecutive bit with the same polarity is detected within the fields were bit stuffing is applied, then a stuff error occurs.

CRC errors: The CRC sequence included in data and remote frames is calculated by the transmitter. When a node receives a data or remote frame it also calculates a CRC sequence. If its CRC sequence does not match the CRC sequence received from the transmitter, then a CRC error has occurred.

Form errors: These are errors where the frame format is violated. Examples include a dominant bit during the CRC delimiter or the ACK delimiter, and a dominant bit during any bit of the EOF.

Acknowledgement errors: These happen whenever a transmitter detects that its ACK slot has not been overridden by a dominant bit, that is, when the transmitter does not detect an acknowledgement of its frame.

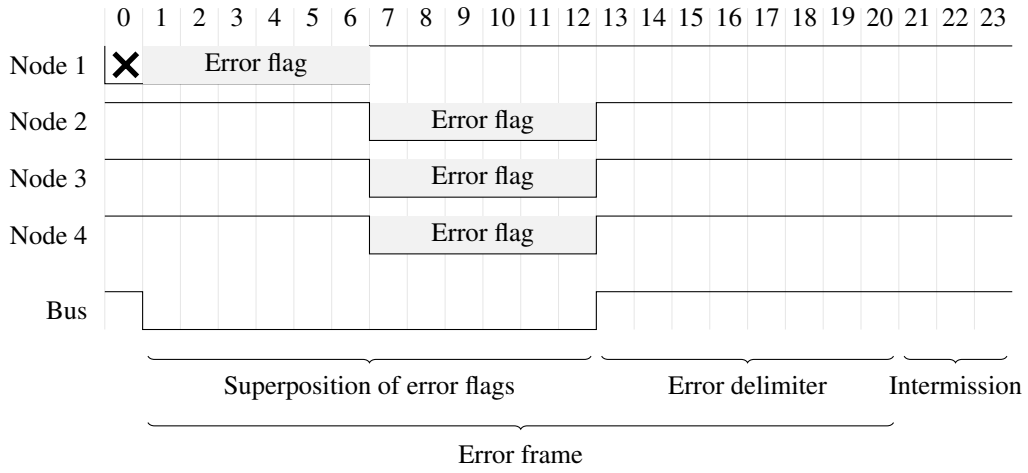

Figure 3.8.: Example of CAN's error-signaling mechanism.

Next we describe the *error-signaling mechanism* while referencing Figure 3.8, which shows an example of how the mechanism works.

When a CAN node detects an error (Node 1 in bit 0) it responds with an *error flag*: a sequence of six consecutive dominant bits starting from the next bit (Node 1, bits 1–6). Because an error flag violates the bit stuffing rule, it is ensured that with the sixth dominant bit of the error flag, at the latest, all nodes will have detected an error (nodes 2–4, bit 6). We therefore say that the error is *globalized*. The result is that the initial error flag triggers the transmission of subsequent error flags from all active nodes that did not detect the initial error (nodes 2–4, bit 7). A node that has sent its error flag (Node 1) must wait for all other error flags to be finished. So, to detect when all nodes have sent their error flags, a node keeps transmitting recessive bits after its error flag (Node 1, bits 7–12) until it monitors a recessive bit on the bus (bit 13). This only happens once all nodes have transmitted their error flags and have started the transmission of their sequence of recessive bits (bit 13). Once the recessive bit is detected on the bus, the nodes start to transmit 7 more recessive bits (nodes 1–4, bits 13–20).

The result seen on the bus after the initial error is a sequence of between 6 and 12 dominant bits—constituted by the superposition of the nodes' error flags—followed by a sequence of 8 recessive bits. The sequence of 8 recessive bits is known as the *error delimiter* and, together with the superposition of the error flags, constitutes an *error frame*. The error frame is then followed by 3 intermission bits, after which the erroneous frame can be retransmitted or a new frame can be transmitted.

3.2.5. Error containment

Without additional mechanisms, the error-signaling described in the previous section could lead a faulty node that keeps detecting local errors—that is, errors only seen by that node—to permanently block any further communication. This would happen because the faulty node would globalize each local error. To avoid this situation, the CAN specification specifies an error-containment mechanism (referred to as fault confinement in the CAN specification [Bosch GmbH, 1991]) based on two error counters for each node: a *Transmission Error Counter* (TEC) and a *Reception Error Counter* (REC).

A node's TEC increases every time the node detects an error while it is transmitting a frame; a node's REC, on the other hand, increases every time the node detects an error while it is receiving a frame. When either of the two error counters equals or exceeds the threshold of 128, the node changes from its normal state, known as the *error-active state*, to the so-called *error-passive state*. The difference between the two states lies in the error flag described in the previous section. A node in the error-active state sends error flags as described previously, that is, consisting of 6 consecutive dominant bits. A node in the error-passive state, in contrast, sends error flags consisting of 6 consecutive *recessive* bits. The former error flag is more precisely called an *active error flag*, whereas the latter is called a *passive error flag*. Note that passive error flags solve the problem of a single receiving node blocking all communication because of permanent local errors. The reason is the following: although the node will continue to transmit error flags, it will no longer be able to interrupt the transmitter because the recessive bits of a passive error flag cannot override any of the bits of the frame being transmitted. Yet, passive error flags transmitted by the transmitter itself are still able to globalize any errors.

If errors persist and the TEC equals or exceeds a second threshold, of value 256, then the node enters the *bus-off* state. In this state a node does not participate in the CAN communication in any way.

To not unfairly penalize a node that has detected sporadic local errors over a long period of time, the TEC and REC counters are not only incremented when errors are detected, but are also decremented when a transmission or reception is successful. This also allows an error passive node to become error active again when the corresponding TEC and REC become less than 128 again. However, when a node becomes bus-off, it cannot recover by decrementing error counters; in that case a recovery can only be accomplished by user request [Voss, 2005].

3.2.6. Overload-signaling

At times a receiving node may need an additional delay before the next frame is transmitted. For this purpose, CAN specifies an *overload-signaling mechanism* which makes use of the 3

intermission bits at the beginning of an interframe space. If a node needs an extra delay, it starts the transmission of an *overload flag* at the first bit of the intermission. The overload flag is, just like an active error flag, a sequence of 6 consecutive dominant bits. When an overload flag is transmitted, the other nodes detect a dominant bit during the intermission and, according to the CAN specification, must react by transmitting an overload flag themselves. Therefore, overload flags are always triggered by the detection of a dominant bit during the intermission, that is, by a form error during intermission.

The remainder of the overload signaling mechanism is equivalent to the error signaling mechanism: a node that has transmitted an overload flag keeps transmitting recessive bits until it monitors a recessive bit on the bus; at this point all nodes have finished the transmission of their overload flags, and they proceed by transmitting 7 additional recessive bits, totaling 8 recessive bits. The result seen on the bus is an *overload frame*, which is constituted of the superposition of the overload flags and the 8 recessive bits transmitted cooperatively by the nodes—the 8 recessive bits being known as the *overload delimiter*.

The overload delimiter is followed again by 3 intermission bits. If a node needs to delay the next frame even more, it can transmit a second overload frame at the first bit of the second intermission. After a second overload frame, however, no further overload frames are allowed.

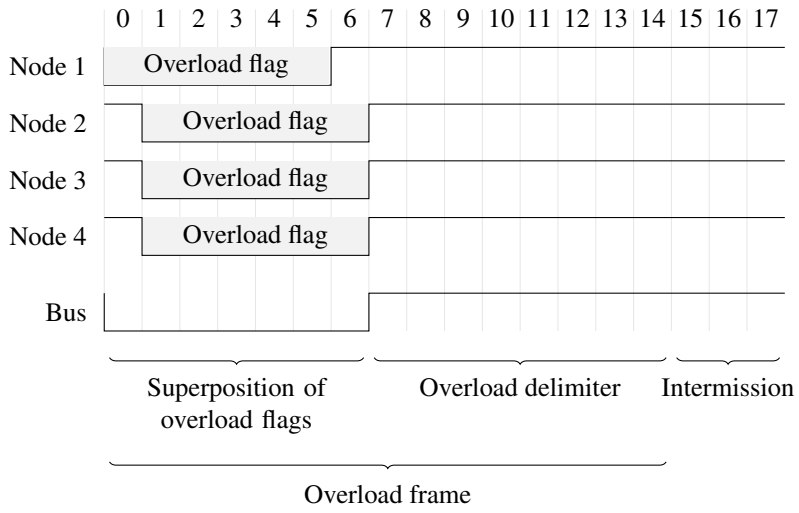

Figure 3.9.: Example of CAN's overload-signaling mechanism. Although not labeled as such in the figure, bits 0, 1, and 2 are the intermission bits following a previous frame. Node 1 overrides the first intermission bit (bit 0) by transmitting an overload flag. This causes a form error in the intermission, which is usually comprised of 3 recessive bits, and causes the other nodes to also transmit overload flags, starting at the second bit of the intermission (bit 1). When all nodes have sent their overload flags, they proceed by transmitting together the overload delimiter. Afterwards, a second intermission follows.

Figure 3.9 shows an example of the overload-signaling mechanism. Although not shown, immediately before bit 0 it is assumed that a frame has been transmitted; bit 0 is therefore the first intermission bit. Node 1 requires more time to process the frame that it has just received and,

thus, it initiates an overload frame by transmitting an overload flag in the first intermission bit (bit 0). This causes nodes 2-4 to detect a form error during the first intermission bit, whereupon they respond by transmitting their own overload flag (bit 1). After all nodes have transmitted their overload flags (bit 7), they transmit the overload delimiter together (bits 7-14), which is then followed by 3 intermission bits (bits 15-17). The result is that Node 1 has more time to process the received frame. Without the overload frame a new frame could have started its transmission in bit 3, that is, after what would have been the intermission without overload signaling. Thanks to the overload signaling, however, the next frame is not transmitted until after bit 17, that is, until after the intermission that follows the overload frame.

3.3. CAN bit rate

All nodes connected to the same CAN bus must transmit and receive at the same *nominal bit rate*. The nominal bit rate is defined as "the number of bits per second transmitted in the absence of resynchronization by an ideal transmitter" [Bosch GmbH, 1991]. Resynchronization is covered in the next section. The *nominal bit time* is the inverse of the nominal bit rate, that is,

$$\text{Nominal Bit Time} = \frac{1}{\text{Nominal Bit Rate}}$$

"The nominal bit time can be thought of as being divided into separate non-overlapping time segments" [Bosch GmbH, 1991]. Figure 3.10 shows the segments that comprise the nominal bit time. These time segments are the following:

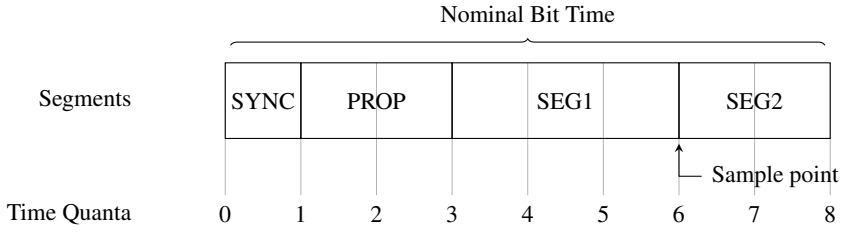

Figure 3.10.: Bit time segments.

Synchronization segment. "This part of the bit time is used to synchronize the various nodes on the bus. An edge [of the bus signal] is expected to lie within this segment" [Bosch GmbH, 1991].

Propagation segment. "This part of the bit time is used to compensate for the physical delay times within the network" [Bosch GmbH, 1991].

Phase segment 1 and phase segment 2. The phase segments "are used to compensate for edge phase errors [and] can be lengthened or shortened by resynchronization" [Bosch GmbH, 1991]. Resynchronization and edge phase errors are described in Section 3.3.1.

The time segments are comprised of integer units of time called *time quanta*, which are fixed units of time derived from the oscillator period. Specifically, a minimum time quantum length is directly derived from the oscillator period and the final time quantum length is obtained by multiplying the minimum time quantum length by a programmable prescaler [Bosch GmbH, 1991], which is "an electronic counting circuit used to reduce a high frequency electrical signal to a lower frequency by integer division" [Wikipedia, 2009] (the prescaler used in CAN to configure the time quantum length is often known as a baud-rate prescaler or a bit-rate prescaler). In mathematical notation, the time quantum length is obtained as follows [Bosch GmbH, 1991]:

$$\mathrm{TQ} = \mathrm{BRP} \cdot \mathrm{TQ}_{\mathrm{min}}$$

where TQ is the time quantum length, BRP is the value of the prescaler, and $\mathrm{TQ}_{\mathrm{min}}$ is the minimum time quantum length. From the specification [Bosch GmbH, 1991] it is not clear how the minimum time quantum length is derived from the oscillator period and not all CAN controllers calculate the time quantum length as stated above. For instance, the CAN controller that we used in our prototype calculates the time quantum length as follows [Microchip, 2006b, Section 26]:

$$\mathrm{TQ} = \frac{2 \cdot (\mathrm{BRP} + 1)}{\mathrm{F}_{\mathrm{CAN}}}$$

where $\mathrm{F}_{\mathrm{CAN}}$ is the frequency derived from the oscillator.

The synchronization segment is always one time quantum long; whereas for the propagation segment and the phase segments the length in time quanta is programmable. Depending on how many time quanta the propagation segment and the phase segments have been programmed to contain, the nominal bit time will contain more or less time quanta. For instance, in Figure 3.10 the propagation segment has been programmed to contain 2 time quanta, phase segment 1 has been programmed to contain 3 time quanta, and phase segment 2 has been programmed to contain 2 time quanta; thus, if the single time quantum of the synchronization segment is included, the nominal bit time in the figure totals 8. Multiplying the number of time quanta contained within the nominal bit time by the time quantum length gives us a value, measured in time units, for the nominal bit time. Continuing with the example, imagine that the time quantum length is 125 ns; this means that the nominal bit time is $8 \cdot 125\,\mathrm{ns} = 1\,\mu\mathrm{s}$. The inverse of this nominal bit time is the configured bit rate. In the example this gives us a bit rate of $\frac{1\mathrm{bit}}{1\,\mu\mathrm{S}} = 1$ megabits per second. Moreover, together the time segments specify when the bus signal is sampled by fixing the *sample point* (also shown in Figure 3.10), which is "the point of time at which the bus level is read and interpreted" [Bosch GmbH, 1991]. The sample point is always located between phase segment 1 and phase segment 2.

3.3.1. Synchronization

In general the nodes in a CAN bus all have their own oscillators. Over time, these oscillators will deviate from each other, that is, although in theory they may all have the same frequency, in reality they have slightly different frequencies and after some time some nodes will have counted more clock ticks than others. Because the length of a time quantum is derived from these oscillators, this means that the length of the time quanta at different nodes will slightly

differ. This may cause problems after some time because the length of the time quanta ultimately determines where the sample point is located for each bit at each node. In the worst case, some nodes may actually sample a previous bit or a subsequent bit instead of the current bit. Moreover, even if the nodes do sample the correct bit, if the sample point is too close to a falling edge or a raising edge of the bus signal, the signal may not be stable yet and the value sampled may be wrong. It is therefore necessary for all nodes to sample the bus at the correct instant of time. This means that the nodes need to synchronize themselves with the currently transmitting node, that is, with the leading transmitter.

There are two mechanisms that the nodes use for synchronization: *hard synchronization* and *resynchronization*.

Hard synchronization only occurs during the recessive to dominant edge of a start of frame (SOF). During a hard synchronization the nodes restart their bit time counters—which keep track of how many bits have passed since the last SOF—thus forcing the recessive to dominant edge of the SOF to lie within the first synchronization segment. This ensures that all nodes initially sample the bus at the correct instant of time.

Resynchronization may be needed after the initial hard synchronization to resynchronize the receiving nodes with the leading transmitter. Specifically, resynchronization is necessary when a receiving node detects that the edge of the received signal falls outside of the synchronization segment. In that case, resynchronization is performed by the nodes by either lengthening phase segment 1 or by shortening phase segment 2. The upper bound for how much the phase segments may be lengthened or shortened is given by a parameter known as the *resynchronization jump width* (SJW), whose value ranges between 1 time quantum, and the minimum of 4 time quanta and the length of phase segment 1, that is, SJW $\in [1, \min(4, \text{SEG1})]$. Whether phase segment 1 is lengthened or phase segment 2 is shortened is determined by the *phase error* of an edge, which gives the position of the edge relative to the synchronization segment measured in time quanta. The phase error, e, of an edge can have three possible signs [Bosch GmbH, 1991]:

- $e = 0$ if the edge lies within the synchronization segment.
- $e > 0$ if the edge lies before the sample point of the current bit, but after the synchronization segment of the same bit.
- $e < 0$ if the edge lies after the sample point of the previous bit, but before the synchronization segment of the current bit.

Figure 3.11 shows an example for each of these cases. In Figure 3.11 (a) the phase error is zero and the falling edge falls within the synchronization segment. In Figure 3.11 (b) the phase error is positive; that is, the falling edge arrives later than expected. Finally, in Figure 3.11 (c) the phase error is negative; that is, the falling edge arrives sooner than expected.

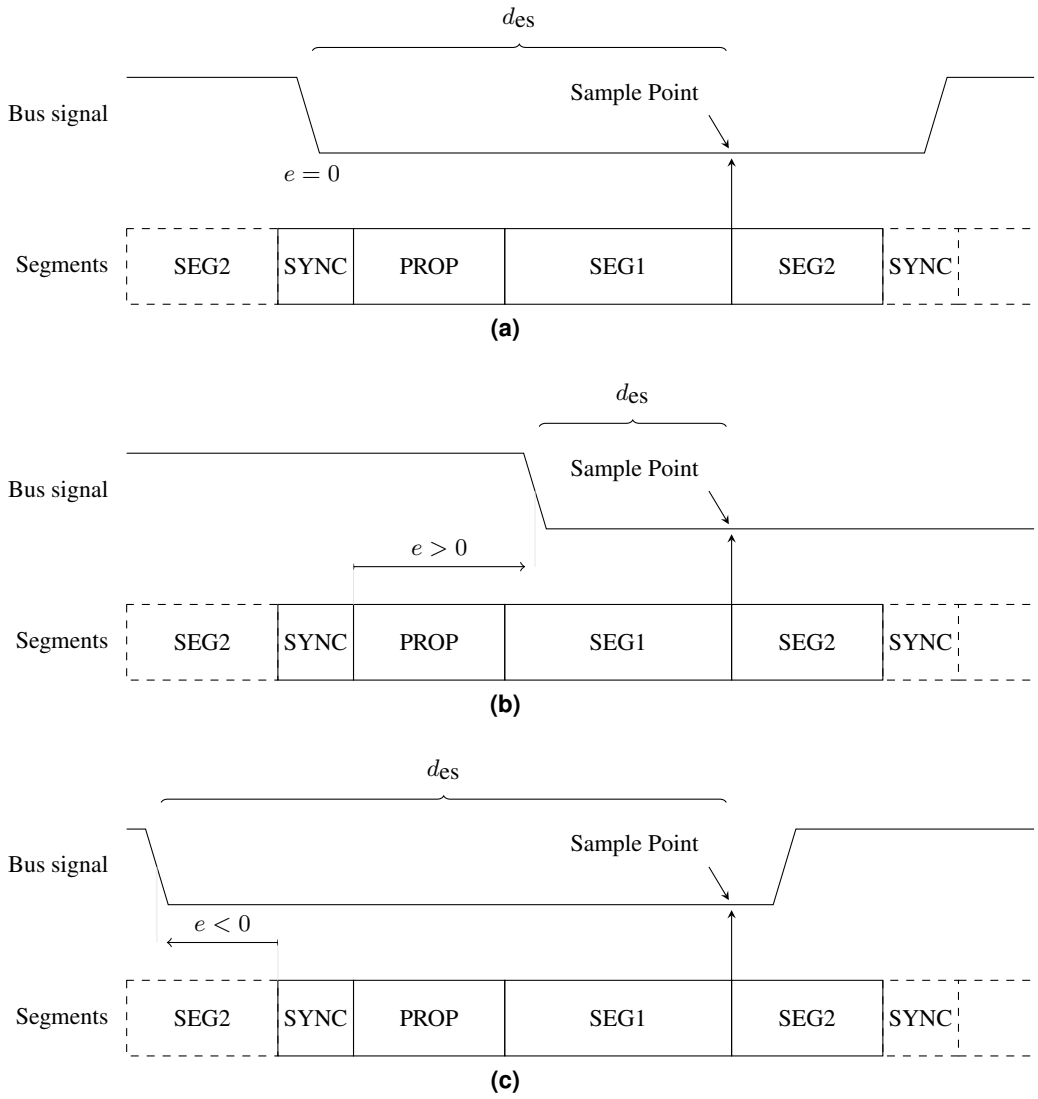

Figure 3.11.: Possible phase errors of an edge in CAN. (a) shows a zero phase error ($e = 0$); (b) shows a positive phase error ($e > 0$); (c) shows a negative phase error ($e < 0$).

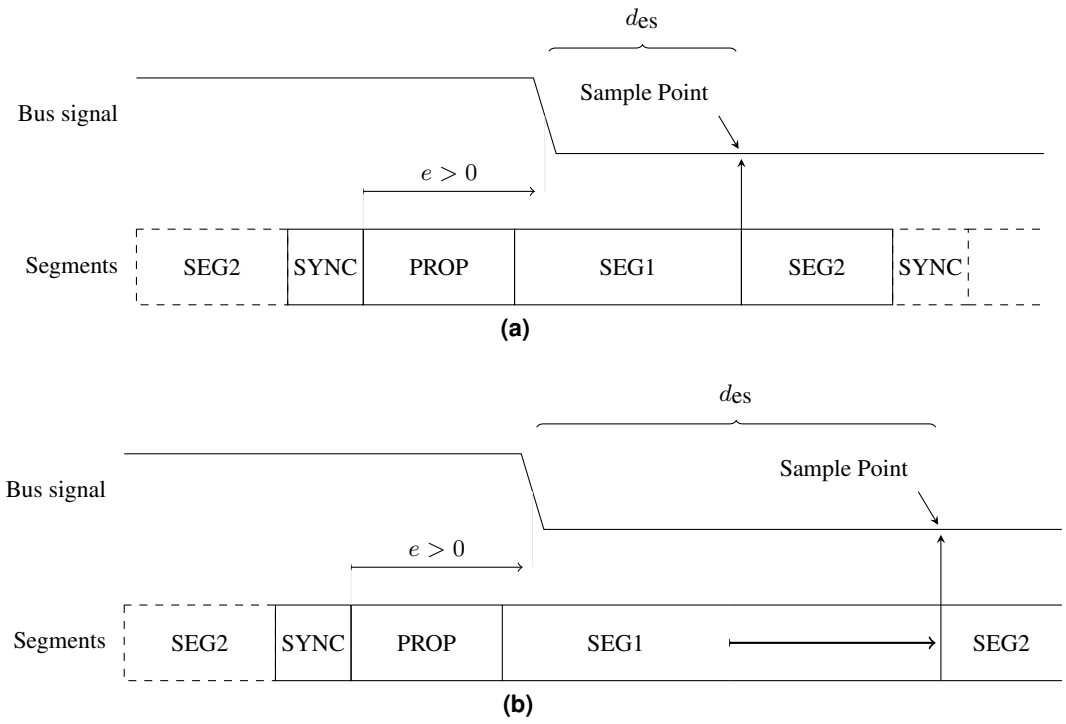

Figure 3.12.: Resynchronization of a positive phase error in CAN. (a) shows a positive phase error without a resynchronization correction; (b) shows a positive phase error corrected by the resynchronization mechanism by lengthening phase segment 1.

Resynchronization is only necessary when the phase error is positive or negative. When the phase error is zero, the node is already synchronized with the leading transmitter and no resynchronization is necessary. The goal of resynchronization is to make all nodes sample the bus at the same distance from their edges. In other words, to ensure that all nodes have the same edge-sample point distance (labeled d_{es} in the figures).

When the phase error is positive, the resynchronization mechanism lengthens phase segment 1. This in turn lengthens the edge sample point distance, thus placing the sample point closer to its correct position for the next bit. Figure 3.12 illustrates this: with positive phase error and no resynchronization the sample point falls too close to the edge (Figure 3.12 (a)); with resynchronization, which lengthens phase segment 1 (Figure 3.12 (b)), the sample point is moved further away from the edge, resulting in an edge sample point distance (d_{es}) which is almost the same as when no phase error occurs.

On the other hand, when the phase error is negative, the resynchronization mechanism shortens phase segment 2. This corrects the sample point's position for the next bit. This is illustrated in Figure 3.13.

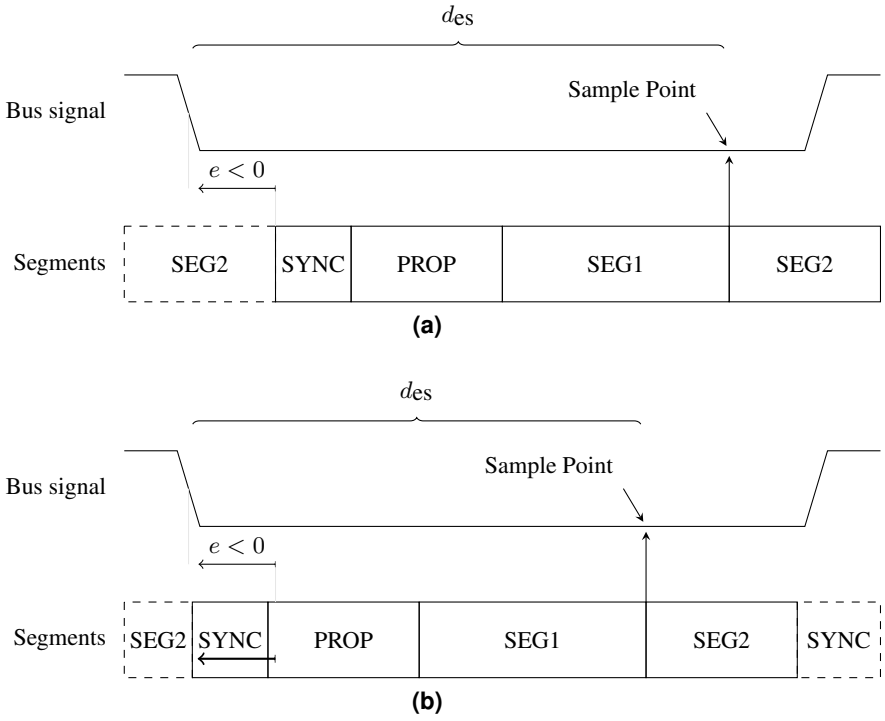

Figure 3.13.: Resynchronization of a negative phase error in CAN. (a) shows a negative phase error without a resynchronization correction; (b) shows a negative phase error corrected by the resynchronization mechanism by shortening phase segment 2 and thus adjusting the sample point position for the next bit.

3.4. Reliability limitations of CAN

As we said previously, CAN is nowadays used for many applications. But despite being widely used, many authors do not consider it suitable for applications which have very demanding reliability requirements. This is due to several limitations that reduce CAN's reliability. Specifically, Pimentel, Proenza, Almeida, Rodíguez-Navas, Barranco, and Ferreira [2008] indicate the following three limitations related to CAN's reliability:

- CAN has limited error containment.
- CAN has limited support for fault tolerance.
- CAN has limited data consistency.

In the next three sections we describe each of these limitations.

3.4.1. Limited error containment

Error containment refers to the capability of a system to restrict the propagation of errors from one subsystem to another. As explained in Section 3.2.5, CAN already has built-in mechanisms

that are used to contain errors at the nodes. These built-in mechanisms are based on the TEC and REC, which are used to switch the state of a CAN controller between error active and error passive and, further, between error passive and bus-off. Unfortunately "the built-in mechanisms are relatively slow to act, depending on the frequency and type of errors" [Pimentel et al., 2008]. Even worse, because of CAN's bus topology, there are many errors which cannot be contained at a single node with CAN's built-in mechanisms [Barranco, Proenza, Rodríguez-Navas, and Almeida, 2004].

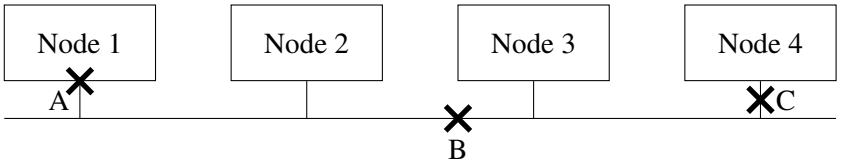

Figure 3.14.: Sample faults in a CAN bus.

As an example, consider Figure 3.14, which shows three points (A, B, and C) where an error might not be contained to a single node. The most obvious point is B, where a partition could prevent the communication between nodes 1 and 2, and nodes 3 and 4. But an error at point A or C might also not be contained to a single node. Point A represents the interface between node 1 and the bus. An error that may occur there and that would not be contained by CAN's built-in mechanisms is, for example, a stuck-at-dominant output of the transceiver of node 1. Such an error would affect the whole bus: any frame sent by any node would be overridden by the constant dominant bits. Similarly point C could affect the whole bus. It represents the stub that connects node 4 to the bus. If, for instance, a physical disruption occurs there, the stub may start to act as an antenna. In that case the resulting bit flipping stream would also affect the whole bus, and CAN's built-in mechanisms could not do anything to contain it.

In general, in a CAN bus an error that cannot be contained can happen on any point of the communication medium. This is so because each node's medium access circuitry, each node's stub to the bus, the bus's main trunk, and, overall, all components attached to the medium, all have direct electrical connections to each other. Furthermore, it is not only errors on the medium that may not be contained, but also errors at the nodes themselves. For instance, a faulty node might repeatedly send without any delay messages with maximum priority; this would lead to a monopolization of the communication medium by a single node and would prevent all other nodes from sending frames.

CAN's error containment is therefore too limited for systems that require high reliability.

3.4.2. Limited support for fault tolerance

As described in Chapter 2, fault tolerance techniques allow a system to deliver a correct service even in the presence of faults, and this is accomplished through error detection and system recovery (refer back to Figure 2.2 on page 12). Error detection is provided on the channel level by CAN through the nodes' capability of detecting bit, stuff, CRC, form, and ACK errors. The specific type of error detection that CAN provides is concurrent error detection, that is, it detects

errors during normal service delivery. Moreover, CAN provides system recovery on the channel level through the retransmission of frames. However, these mechanisms can only deal with faults at the channel level and cannot deal with any faults occurring at higher levels, such as faults at CAN controllers and microcontrollers. Moreover, CAN does not have any mechanisms designed to facilitate the addition of redundancy required for fault tolerance above the channel level.

3.4.3. Limited data consistency

CAN has some specific protocol characteristics which may lead to data inconsistencies, that is, to scenarios where not all nodes agree whether a frame should be accepted or not. This can lead to *inconsistent message omissions* or to *inconsistent message duplicates*. Inconsistent message omissions are scenarios where some nodes receive a specific message while others do not; whereas inconsistent message duplicates are scenarios where a given message is received by some nodes twice while it is received only once by other nodes [Pimentel et al., 2008]. Because data inconsistencies may impede the nodes to reach a consensus, they can prevent the correct service of a distributed system comprised of these nodes; hence, data inconsistencies can make the system unreliable.

There are several ways in which data inconsistencies can arise in CAN. One way is for a receiving node to be in the error-passive state and to be the only one to detect an error in the received frame. Because it has detected an error in the frame, the error-passive node will not accept that frame. If the node were in the error-active state, it would send an active error flag and thereby force the retransmission of the frame; after the retransmission, all nodes would have accepted the frame. However, because it is in the error-passive state, the node will not be able to force the retransmission of the frame and will therefore be the only one that will not have received a copy of the frame. The result is an inconsistent message omission [Proenza and Miro-Julia, 2000].

Other reasons why data inconsistencies can occur in CAN are due to a special behavior of CAN in the last bits of a frame. So, before we can explain the other data inconsistency scenarios, we must first explain this behavior of CAN.

The special behavior of CAN in the last bits of a frame is illustrated in Figure 3.15. Figure 3.15 (a) shows what would happen if a set of receivers, labeled X, detect an error in the last bit of a frame and there was no special rule for the last bit of a frame. They would reject the received frame and start to signal an error flag during the first intermission bit. This, however, would be interpreted by the transmitter and by other receiving nodes that had not detected an error, labeled Y in the figure, as an overload flag. Thus, these other receivers would accept the frame and the transmitter would not carry out a retransmission of the frame. This would lead to a situation where the nodes labeled X would not have accepted the frame while the nodes labeled Y would have accepted it. In other words, an inconsistent message omission would have occurred. To avoid this, CAN has a special rule when an error is detected by a receiver in the last bit of a frame. This rule is illustrated in Figure 3.15 (b). When a set of receivers, labeled X in the figure, detects an error in the last bit of an EOF, they must simply accept the received frame. Thanks to this rule, all receivers do accept the frame and there is no longer an inconsistent message omission.

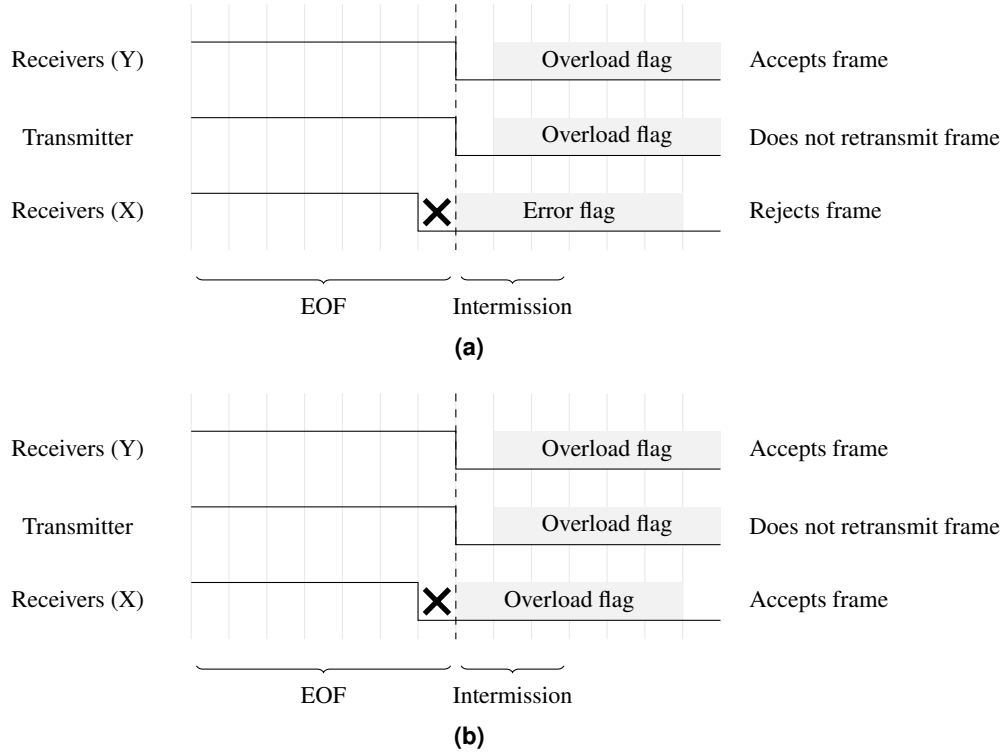

Figure 3.15.: CAN's last bit rule. (a) shows what would happen without the last bit rule in CAN. (b) shows the last bit rule in CAN: to ensure that the receivers labeled X do get a copy of the frame, CAN's last bit rule states that if a receiver detects an error in the last bit of a frame, it must accept the frame.

Unfortunately, the attempt to solve inconsistent message omissions with the special behavior in the last bit of a frame introduces new data inconsistency scenarios.

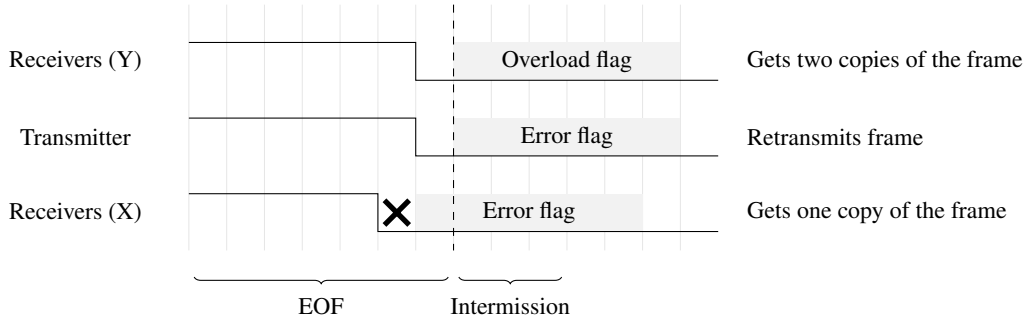

Figure 3.16.: Inconsistent message duplication scenario discussed by Rufino et al. [1998] and Proenza and Miro-Julia [2000].

First, consider what would happen if a set of receivers detect an error in the next-to-last bit of a frame while another set of receivers and the transmitter do not detect such an error. This is illustrated in Figure 3.16. The set of receivers labeled X are the ones that detect the error in the next-to-last bit; whereas the set of receivers labeled Y are the ones that do not detect the error. The set of receivers X respond to the error they detect by transmitting an error flag, whose transmission starts during the last bit of the EOF. This means that the set of receivers Y and the transmitter detect the first dominant bit of the error flag during the last bit of the EOF. The transmitter responds to this dominant bit by transmitting its own error flag and by carrying out a retransmission. For the set of receivers Y, however, the special rule for the last bit of a frame applies; thus, they accept the frame and transmit an overload flag. This means that the Y set will have accepted the first transmission of the frame while the X set will not have accepted it and, after the transmitter's retransmission, the Y set will have accepted two copies of the frame while the X set will have accepted only one. The result is an inconsistent message duplication [Proenza and Miro-Julia, 2000; Rufino et al., 1998]. Fortunately, inconsistent message duplications are not as bad as inconsistent message omissions because they can be dealt with by following a set of recommendations, such as avoiding the transmission of frames that toggle the state of the receivers or that convey information that is relative to the data received in other frames [Zeltwanger, 1998, as cited by Proenza and Miro-Julia 2000].

Next, consider the scenario that Rufino et al. [1998] have identified. The scenario is illustrated in Figure 3.17. The situation is essentially the same as in the scenario of Figure 3.16, with the key difference that the transmitter is not able to carry out a retransmission of the frame because it suffers a failure shortly after the first transmission of the frame. This leads to a situation in which the set of receivers X will not have received any copy of the frame, whereas the set of receivers Y will have received a copy. In other words, this scenario leads to an inconsistent message omission.

Finally, another data inconsistency scenario is the one that Proenza and Miro-Julia [2000] have identified. This scenario is illustrated in Figure 3.18. As in the scenarios of figures 3.16 and 3.17, an error is detected in the next-to-last bit of the EOF by a set of receivers labeled X.

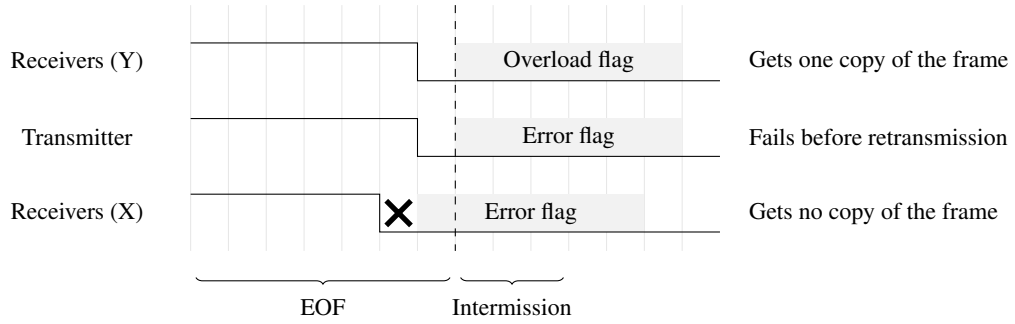

Figure 3.17.: Inconsistent message omission scenario identified by Rufino et al. [1998].

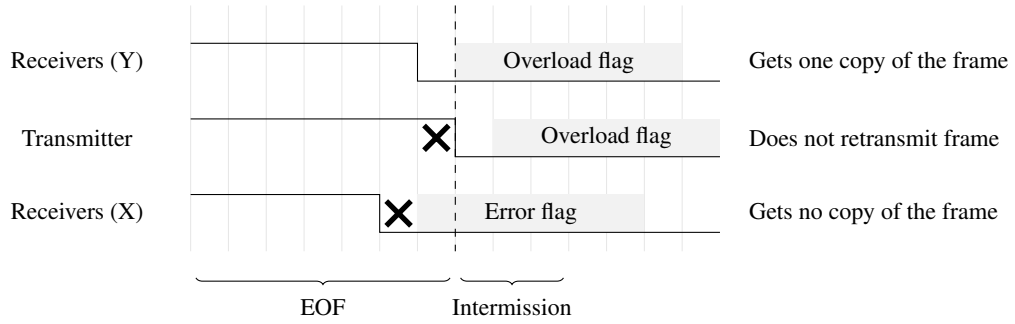

Figure 3.18.: Inconsistent message omission scenario identified by Proenza and Miro-Julia [2000].

These receivers then start the transmission of an error flag in the next bit so that the first bit of the error flag coincides with the last bit of the EOF. This causes another set of receivers, labeled Y, to detect a dominant bit in the last bit of the EOF. Following CAN's rule for the last bit, these other receivers accept the transmitted frame and transmit an overload flag. The transmitter, on the other hand, does not detect the dominant bit in the last bit of the EOF because of a second disturbance of the medium that only affects this node. Instead, the first dominant bit belonging to an error flag that the transmitter detects coincides with the first intermission bit. This means that the transmitter interprets that dominant bit as the start of an overload flag and, thus, transmits its own overload flag without carrying out any retransmission afterwards. The end result is an inconsistent message omission: the set of receivers X gets no copy of the frame while the set of receivers Y gets one copy.

4. CANcentrate

This chapter introduces *CANcentrate*, an active star topology to improve the error containment in CAN networks. It is the precursor of ReCANcentrate and has been designed and implemented as a prototype by Barranco et al. [2006b]. Because ReCANcentrate builds upon CANcentrate, it is very useful to understand CANcentrate first. The purpose of this chapter is to provide this understanding.

4.1. Fault model for CAN and CANcentrate

As we said in Chapter 2, when designing a fault-tolerant system, one has to make a few assumptions about the possible failure modes, that is, the different ways in which failures can manifest. This is known as a fault model. Barranco [2010] classified the faults that can occur in a CAN network into two main categories.

The first category are *syntactic faults*, that is, "faults that generate errors that corrupt the bit values that are broadcast through the medium". These faults can further be classified into two subcategories:

Stuck-at faults: these occur "whenever a given node or a component of the medium is damaged and issues a constant bit value" [Barranco, 2010]. If the constant bit value is dominant, the fault is more precisely called a stuck-at-dominant fault, and if the constant bit value is recessive, it is called more precisely a stuck-at-recessive fault. Note that because of CAN's wired-AND, a stuck-at-recessive node cannot disturb the whole network—recessive bits issued by that node cannot override bits from other nodes. However, if a stuck-at-recessive fault affects the communication medium itself, because of a short-circuit of the medium for instance, then the stuck-at-recessive fault does affect the whole network. Stuck-at-dominant faults, on the other hand, affect the whole network wherever they happen—again, because of the wired-AND, which leads dominant bits to override any recessive bits.

Bit-flipping faults: these occur whenever a node or a component issues random bits or randomly changes the values of a passing bit stream. They may be caused, for instance, by a bad welding.

The second category are *semantic faults*. These include, for instance, faults that lead a node to transmit frames too early or too late, which Barranco claims is a typical problem of communication systems. Barranco highlights two types of semantic faults:

Babbling-idiot faults: these occur "whenever a node sends messages that are erroneous in the time domain" [Barranco, 2008], thereby preventing other nodes from transmitting their

frames. An example has already been mentioned in the previous chapter (Section 3.4.1): a node that monopolizes the communication medium and starves other nodes by repeatedly transmitting a high priority message without delay. Note that what constitutes a babbling-idiot fault is application dependent. For instance, a long stream of high priority messages from a single node may not constitute a babbling-idiot fault, but be appropriate for a particular application.

Network partition faults: these occur "whenever the network is broken into several subnetworks, which are called *network partitions*" [Barranco, 2010]. A network partition fault prevents nodes on different resulting network partitions to communicate with each other.

The CAN protocol can only deal with syntactic faults [Barranco, 2010]. This, together with the reliability limitations of CAN (described in the previous chapter), motivated the design of CANcentrate.

The fault model for CANcentrate includes both syntactic and semantic faults. However, because the work of Barranco [2010] focuses on application *in*dependent solutions, the treatment of semantic faults that require information specific to the application is currently not considered for CANcentrate. Thus, the only semantic faults that CANcentrate deals with are network partitions.

In summary, the CANcentrate fault model includes stuck-at, network partition, and bit-flipping faults. Moreover, because CANcentrate has no replicated hubs, it also assumes that the hub cannot fail.

4.2. CANcentrate's architecture

CANcentrate's basic architecture is depicted in Figure 4.1. It includes a single hub that implements fault-tolerance techniques and to which each node is connected by a downlink and an uplink. A downlink is used by a node to receive the signals sent by other nodes through the hub or to receive the signals sent by the hub itself; an uplink is used by a node to transmit signals through the hub to other nodes. Note that since each node has its own dedicated link to the hub and the hub is assumed to never fail, there is no way that a network partition fault can occur.

Each downlink and uplink is comprised of both a CANH and a CANL wire. The differential voltage between these CANH and CANL wires cannot directly be used by the internal components of the nodes or of the hub (we assume that the CAN controllers used by the nodes do not implement the Physical CAN layer). Transceivers are therefore needed to convert the differential voltage to logic levels usable by the internal components. Specifically, a transceiver is needed at each end of each uplink and downlink. That means that each node has two CAN transceivers, one for the uplink and one for the downlink, and that each port of the hub, where the up- and downlink of a node are plugged in, also has two transceivers.

The physical setup of a node is shown in Figure 4.2. The node's microcontroller is connected to a CAN controller (labeled CAN). The CAN controller in turn is connected to the two CAN transceivers (labeled Txrx): the CAN controller's transmission pin (Tx) is connected to the transmit data input (TxD) pin of the uplink's transceiver, and the CAN controller's receive pin (Rx) is connected to the receive data output (RxD) pin of the downlink's transceiver. The receive

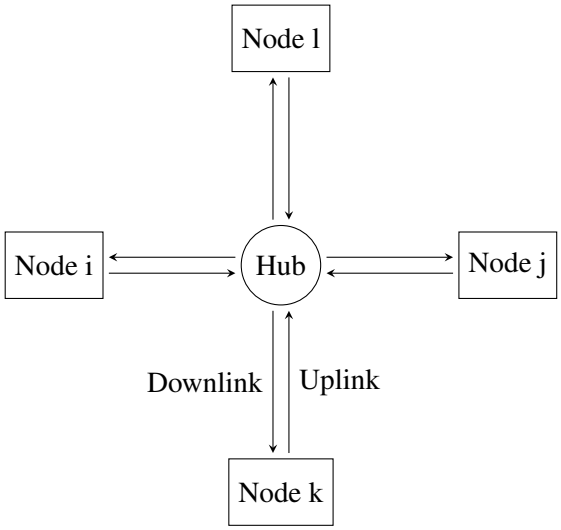

Figure 4.1.: CANcentrate's architecture for four nodes. Each node is connected to the central hub by means of a link comprised of a downlink and an uplink. (The figure is based on a figure by Barranco et al. [2006b].)

pin (RxD) of the uplink's transceiver and the transmit pin (TxD) of the downlink's transceiver are left unconnected (Barranco et al. [2006b] connect a logical 1 value to the transmit (TxD) pin of the downlink's transceiver to force a recessive value on the downlink, however, this is unnecessary—at least with the PCA82C250 transceivers we used in our prototype—because the transceivers do output recessive values by default [Philips Semiconductors, 1996]).

Figure 4.3 shows the internal structure of a CANcentrate hub. At the hub's end of each uplink and downlink is the hub's *Input/Output Module*. As shown in the lower half of the figure, the Input/Output Module is made up of a series of transceivers: for each uplink and for each downlink there is a transceiver (Tr) in the hub's Input/Output Module. The connection of these transceivers is analogous to how a node's transceivers are connected: receiving transceivers are connected to the rest of the hub by their receive data output pins, and transmitting receivers are connected to the rest of the hub by their transmit data input pins. The remaining transmit pins in receiving transceivers are connected to a logical 1 (although they may be left unconnected as in a node's downlink transceiver), and the remaining receive pins in transmitting transceivers are left unconnected.

The wired-AND functionality of a CAN bus is performed at the hub's *Coupler Module*, which is shown in the upper left corner of Figure 4.3. This module couples the signals from the uplink transceivers of the Input/Output Module (B_1, B_2, ..., B_n) in an internal AND gate, that is, it couples the signals that originate at the nodes. The output of the AND gate (B_0) is then broadcast to the nodes through the downlink transceivers of the Input/Output module. The frame that results from the coupling is called the *resultant frame*.

Figure 4.3 also shows several OR gates in the coupler module. Specifically, there is an OR gate between each uplink transceiver and the AND gate. A signal coming from an uplink transceiver

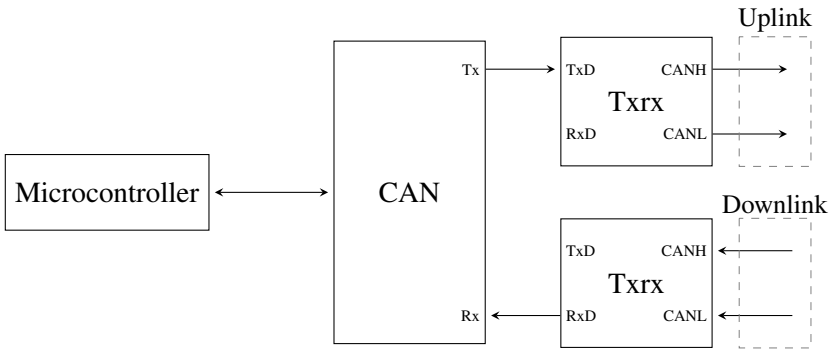

Figure 4.2.: CANcentrate node architecture. (The figure is based on a figure by Barranco et al. [2006b].)

therefore first enters a corresponding OR gate before it can reach the AND gate. The purpose of these OR gates is to allow the isolation of faulty uplinks. This is accomplished through the second input to each OR gate, which originates at an *enabling/disabling unit* of the *Fault-Treatment Module*. As long as this second input is a logical 0, the signal from the corresponding uplink reaches the AND gate without having its logical value altered: if the uplink signal is a logical 1, it is still a 1 after being ORed with the 0 of the enabling/disabling unit; analogously, if the uplink signal is a logical 0, it is still a 0 after being ORed with the enabling/disabling unit's 0. However, if the corresponding enabling/disabling unit outputs a logical 1, then the corresponding uplink's signal is forced to become a logical 1 before entering the AND gate. The enabling/disabling units can therefore force the signals from specific nodes to become a constant sequence of 1's—which are equivalent to recessive values. A node whose contribution is forced to be recessive values is effectively isolated: it cannot send any signals to the other nodes, and the forced sequence of 1's is equivalent to the non-existence of the node. So, all that is needed to isolate a node's erroneous signal is that the corresponding enabling/disabling unit outputs a 1.

But how does an enabling/disabling unit know whether the signals coming from its corresponding uplink are erroneous and should therefore be isolated? For this we have to look at the whole hub. First, notice that the hub can discriminate between the contribution of each node because each node's contribution arrives at the hub by its own separate uplink. Next, consider that whether a particular bit value sent by a node is erroneous or not depends on what that node was expected to transmit: if a node transmits a dominant bit when a recessive bit is expected, the bit is erroneous; similarly, if a node transmits a recessive bit when a dominant bit is expected, the bit is also erroneous. Obviously, one cannot always know what bit value each node should transmit at each given instant—for instance, generally one cannot predict what the leading transmitter should send in its data field. Nevertheless, in many instances one can predict the correct contribution of a node. A few examples are the following:

- While there is a leading transmitter, no other node should send dominant bits, except during the ACK slot.
- Receiving nodes should send a dominant bit during the ACK slot.

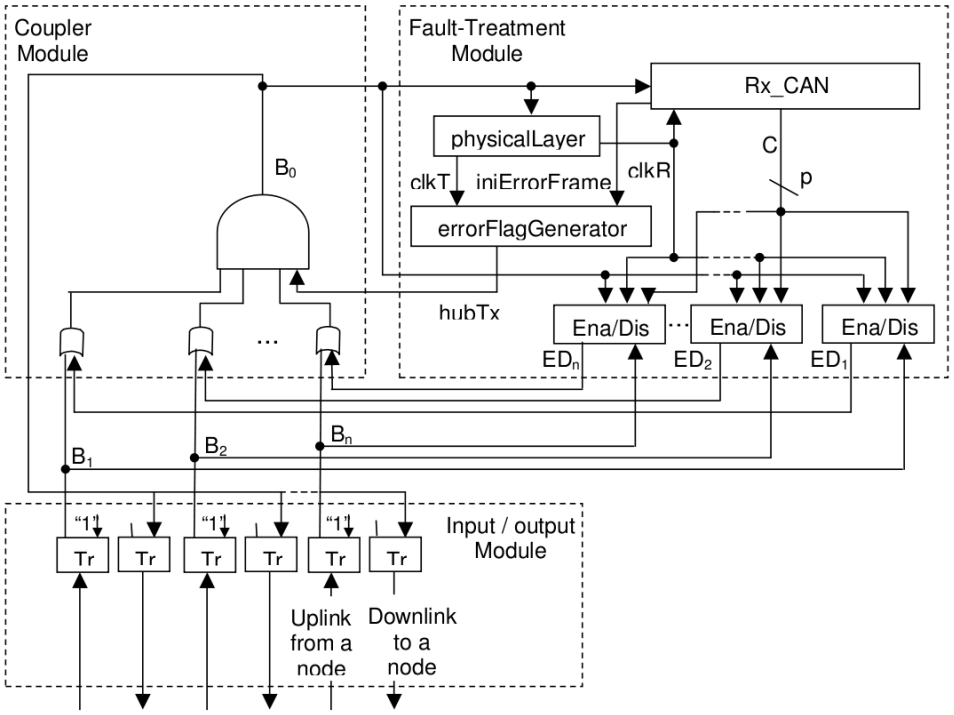

Figure 4.3.: Internal structure of a CANcentrate hub. (Reprinted from a technical report by Barranco [2008] with his permission.)

- When a stuff error occurs in the resultant frame, all nodes should send an error flag.
- When a CRC error occurs, all receiving nodes should send an error flag.

In order for the hub to predict a node's correct contribution in such instances, it needs to know what Barranco et al. [2006b] call the *current state of the resultant frame*. This current state of the resultant frame "represents what all nodes are supposed to have received from the hub until this moment" and, therefore, "permits to identify which is the meaning of the bit of the resultant frame that is currently being broadcasted to all ports, as well as to forecast which should be the proper contribution of each node for the following bit" [Barranco et al., 2006b].

To maintain the current state of the resultant frame, the hub needs to be synchronized at the bit level with the nodes; otherwise it would not know when to sample the nodes' signals to determine their current bit value. For this, the hub uses the *Physical Layer Module*, which is shown in Figure 4.3 and which is part of the Fault-Treatment Module (we think the name Physical Layer Module is poorly chosen because it deals with bit timing and synchronization, which according to the CAN specification [Bosch GmbH, 1991] are part of the Data Link Layer and not part of the Physical Layer).

Moreover, to maintain the current state of the resultant frame, the hub also needs to know which part of a frame, that is, which frame field, is currently being transmitted. Barranco

et al. [2006b] call this *frame-level synchronization*. Frame-level synchronization is done by the *Rx_CAN Module*, which is part of the Fault-Treatment Module, by observing the coupled signal B_0. As a result of this synchronization, the Rx_CAN Module generates a set of signals, labeled C in Figure 4.3, which, together with the coupled signal B_0, describes the current state of the resultant frame. The enabling/disabling units then use this description of the current state of the resultant frame to determine whether their corresponding uplink has suffered an error. If it has, a series of error counters inside the corresponding enabling/disabling unit are incremented accordingly. If these counters reach a given threshold, the corresponding uplink is diagnosed as faulty and isolated.

Finally, note that when errors are present, the hub can generally no longer know to what frame field the currently transmitted bit belongs—just think of an error in the DLC field, which would lead the hub to no longer know when the data field ends. Therefore, when the hub detects an error, it globalizes the error by transmitting its own active error flag. This aborts the frame being transmitted and forces a resynchronization between the nodes and the hub. The transmission of an active error flag is the purpose of the *error flag generator module*, shown in the Fault-Treatment Module of Figure 4.3.

5. ReCANcentrate

In the fault model for CANcentrate, Barranco [2010] assumed that the CANcentrate hub would not fail. Of course in reality the hub may fail, leading to an overall failure of the system. Barranco et al. [2006b] acknowledged that this is an obvious drawback of CANcentrate, which, however, can be faced by placing the hub in a well-protected area of the physical system, by investing in its quality, or—and this is where ReCANcentrate comes in—by adopting a replicated star topology with more than one hub.

In this chapter we cover ReCANcentrate in more detail than in the introduction: we give the fault model used by Barranco [2010] for ReCANcentrate, we revisit the ReCANcentrate architecture in more detail, we summarize the internal structure of a ReCANcentrate hub, we describe the architecture and media management of the ReCANcentrate nodes, and we describe the first ReCANcentrate prototype, which Barranco et al. implemented before this project.

5.1. Fault model for ReCANcentrate

The fault model that Barranco [2010] defined for ReCANcentrate is very similar to the fault model defined for CANcentrate (see Section 4.1). Like the fault model for CANcentrate, the one for ReCANcentrate includes syntactic faults—that is, stuck-at and bit-flipping faults—and network partition faults. However, there are some differences.

First, in the fault model for ReCANcentrate it is no longer assumed that a hub cannot suffer a failure; instead, it is assumed that hubs can fail as long as at least one hub remains non-faulty. Regarding the failure modes of the hubs, they are assumed to only suffer syntactic faults, that is, faults that generate stuck-at or bit-flipping bits, and not semantic faults. This is reasonable because, as will be clear later in this chapter, hubs cannot build or store frames.

Second, as opposed to CANcentrate's simplex star topology, ReCANcentrate's replicated star topology does not inherently prevent network partition faults. In a replicated star topology nodes are connected to more than one hub. If links between hubs and nodes fail, the resulting combination of remaining non-faulty links may lead the nodes to no longer agree which nodes are participating in the communication. For instance, consider Figure 5.1, where the link between hub 1 and node B, and the link between hub 2 and node C have suffered a failure. Node A can still communicate with both nodes B and C; whereas nodes B and C can only communicate with node A, but cannot communicate with each other. The next section describes how network partition faults are avoided.

Finally, the fault model for ReCANcentrate also includes the possibility of a CAN controller *crash*. When this occurs, the CAN controller stops notifying its node about anything and is, from the network's point of view, stuck-at-recessive. Note that this possibility was not included in the fault model for CANcentrate because there each node only has a single CAN controller. In

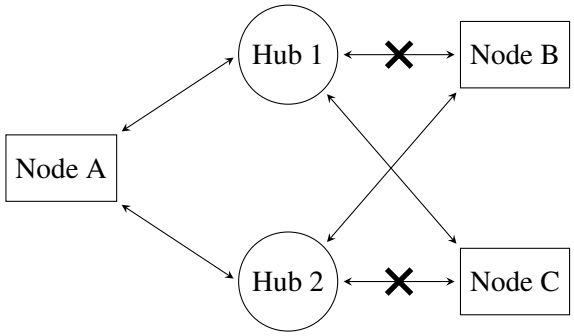

Figure 5.1.: Example of a *network partition* fault. (The figure is based on a figure by Barranco [2010].)

contrast, as described in Section 5.3, in ReCANcentrate each node has a CAN controller for each of its links. Thus, in ReCANcentrate it should be possible to tolerate the crash of one controller per node.

In summary, ReCANcentrate has been designed to contain errors at links, hubs, and nodes that manifest themselves as stuck-at or bit-flipping faults in the channel. Moreover, it has been designed to tolerate up to one CAN controller crash per node and the fault of one link per node, of one hub, and of all but one interlink.

5.2. ReCANcentrate hub architecture

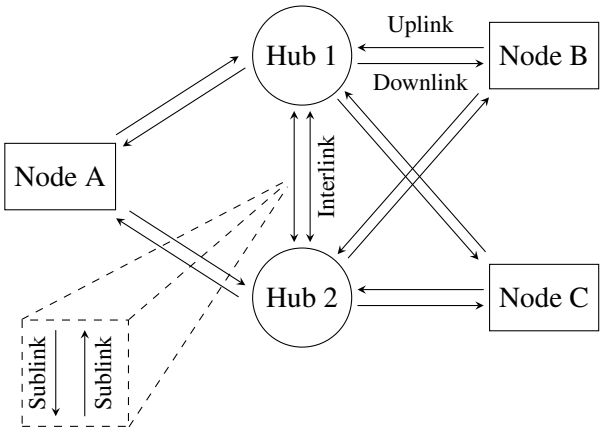

Figure 5.2.: ReCANcentrate's architecture. (The figure is based on a figure by Barranco et al. [2006a].)

Figure 5.2 shows ReCANcentrate's basic architecture. Comparing it with the CANcentrate architecture of the previous chapter, the notable difference is that there are now two hubs instead of just one. Each node is connected to each of these two hubs by an uplink and a downlink. Like

in CANcentrate, an uplink carries the signals from a node to a hub, and a downlink carries the signals in the other direction, from a hub to the node. The two hubs are connected to each other by replicated links called *interlinks*, which are comprised each of two *sublinks*. One sublink carries signals from one hub to the second hub, and the other sublink carries signals in the opposite direction, from the second hub to the first hub. The signals that one hub transmits to the other are the coupling of the signals it receives from its uplinks—just like in CANcentrate, the coupling is done in an internal AND gate. For a given hub, this coupled signal is referred to by Barranco et al. [2005a] as the *contribution* of that hub.

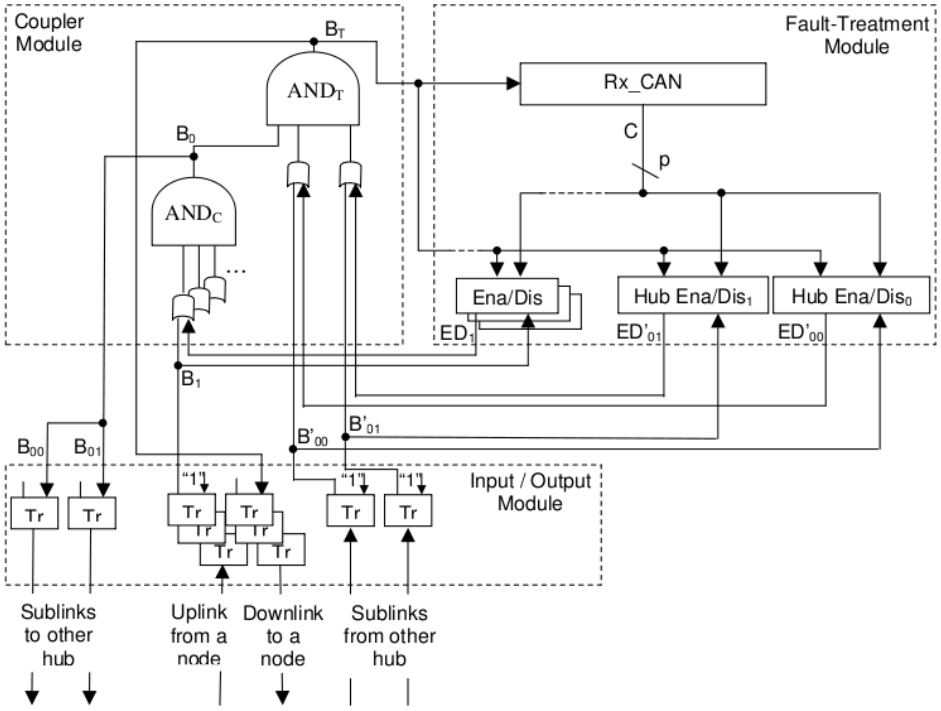

Figure 5.3.: Internal structure of a ReCANcentrate hub. (Reprinted from a paper by Barranco et al. [2005a] with Barranco's permission.)

Figure 5.3 shows the internal structure of a ReCANcentrate hub. The sublinks are each a pair of CANH and CANL wires, just like the up- and downlinks; therefore the Input/Output module of a ReCANcentrate hub not only includes a transceiver for each uplink and downlink, but also includes a transceiver for each sublink (see the bottom half of Figure 5.3). A hub couples the contribution it receives from the other hub through the incoming sublinks with its own contribution in a second AND gate (AND_T in Figure 5.3). The result of this second AND-coupling (B_T) is what a ReCANcentrate hub broadcasts to its nodes using its downlinks. The frame that results from this second AND-coupling is called the *resultant frame* [Barranco et al., 2005a].

Because each of the hubs performs this second AND-coupling within a fraction of the CAN

bit time, they create a single logical broadcast domain, that is, both hubs transmit the exact same value, bit by bit, through their downlinks. The coupling of both hubs' contributions has the following consequences [Barranco et al., 2005a]:

- Network partition faults are prevented: regardless of the hub or hubs a node is connected to, it receives the same traffic as all the other nodes that are connected to at least one hub.
- A node can communicate as long as one of its two uplinks and one of its two downlinks is non-faulty, even if it transmits through the uplink to one hub and receives through the downlink from the other hub (note that this is only true in principle, but that the current design of the ReCANcentrate nodes, described in Section 5.3, requires both an uplink and a downlink of the same link to be non-faulty to allow a node to communicate).
- In replicated communication systems a common difficulty is detecting when frames received at different replicas of the channel are actually copies of the same frame. It is also a common difficulty to detect when a given frame has been received on one channel, but omitted on another. In ReCANcentrate this is easily solved because of the coupling of both hubs' contributions: for nodes connected to both hubs "duplicated frames are always expected in each reception, whereas an omission can be easily detected by checking, at the reception of each frame, that two copies of the same frame are effectively received from both stars" [Barranco et al., 2005a]. This is covered in more detail in Section 5.3.

Regarding fault isolation, just like a CANcentrate hub, a ReCANcentrate hub uses enabling/disabling units together with OR gates to isolate faulty uplinks. The same strategy is also used to isolate faulty sublinks—or a faulty hub, which will manifest itself as faulty sublinks from the isolating hub's point of view. That is, a ReCANcentrate hub has additional *hub enabling/disabling units* and additional OR gates for the incoming sublinks. The additional OR gates are placed between the transceivers of the incoming sublinks and the second AND gate (AND_T) (see Figure 5.3). Similar to the enabling/disabling units, the hub enabling/disabling units decide whether to isolate their sublinks based on the signals they receive from the sublink's transceivers (B'_{00} and B'_{01} in Figure 5.3), based on the resultant frame (B_T), and based on the set of signals (C) they receive from the Rx_CAN module.

The Rx_CAN module of a ReCANcentrate hub has the same function as the Rx_CAN module of a CANcentrate hub: it monitors the coupled signal that is broadcasted to the nodes (B_T in the case of ReCANcentrate); maintains frame level synchronization, that is, it identifies to what part of a frame each transmitted bit belongs; and it generates a set of signals (C), which, together with the coupled signal (B_T), describes the current state of the resultant frame.

5.3. ReCANcentrate nodes

With respect to the nodes in CANcentrate, the nodes in ReCANcentrate have to be modified in order to be connected to two hubs and to properly manage the replicated media provided by the hubs. Figure 5.4 shows the architecture of a ReCANcentrate node. It is constituted by commercial-off-the-shelf (COTS) components only: a ReCANcentrate node has one microcontroller, two CAN controllers, and four transceivers. Each of these two CAN controllers is

connected to a different hub through its own pair of transceivers: one of the transceivers is used to translate the differential voltage from a downlink to a logical value a CAN controller can understand, and the other is used to translate the logical value of a CAN controller to a differential voltage on the uplink.

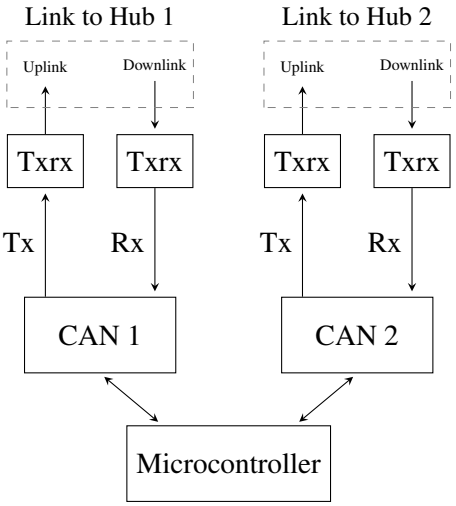

Figure 5.4.: ReCANcentrate node architecture. Each CAN controller (CAN 1 and CAN 2) is connected to only one hub, and this is done by means of two transceivers (Txrx) per CAN controller, one for the uplink and one for the downlink.

Next we describe the media management strategy used by the nodes, that is, how the nodes transmit and receive through the replicated star while tolerating faults.

5.3.1. Media management in the absence of faults

ReCANcentrate nodes use both controllers to receive frames, but only one of the controllers to transmit frames. The controller that is both used to transmit and receive frames is called the *transmission controller*, whereas the controller that is only used to receive frames is called the *non-transmission controller*.

As we said in Section 5.2, the hubs couple their contributions and create a single logical broadcast domain. Therefore, when a frame is successfully exchanged through the network, that is, when a *delivery event* occurs, each node expects that its two CAN controllers nearly simultaneously notify of that event. The nodes' media management strategy takes advantage of this fact. In the absence of faults, a node manages transmissions and receptions as follows. First, if the node successfully transmits a frame, the transmission controller notifies the transmission of this frame and the non-transmission controller notifies the reception of this frame; in that case, all the node needs to do is accept the notification of the transmission as valid and release the reception buffer of the non-transmission controller to empty it for the next *delivery event*, that is, for the next frame exchange. 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, both controllers

will contain the same frame due to ReCANcentrate's single broadcast domain. Therefore, when both controllers notify a reception, the node's microcontroller loads the frame from either CAN controller and, subsequently, releases the reception buffers of both controllers so that they are empty for the next delivery event.

5.3.2. Media management in the presence of faults

The starting point for what faults a node must deal with is the ReCANcentrate fault model. As we said in Section 5.1, it includes controller crashes as well as network partition faults, syntactic faults (i.e. bit-flipping or stuck-at), and hub faults. Nodes must deal with syntactic and hub faults—both of which manifest, from a node's point of view, as syntactic faults in an uplink or downlink. Moreover, they must deal with controller crashes. Nodes, however, do not have to deal with network partition faults because these are inherently solved in ReCANcentrate due to the hub coupling. Finally, the nodes have also been designed to avoid the inconsistent message omission scenario identified by Rufino et al. and, in some situations, to avoid the inconsistent message omission scenario identified by Proenza and Miro-Julia (see Section 6.4).

If a fault occurs in the communication channel, it generates errors that block the communication for all nodes. This is so because since ReCANcentrate enforces a single broadcast domain, the error is globalized and all CAN controllers signal error frames as long as the fault continues to generate errors. If the fault is temporary, the communication is reestablished as soon as the fault becomes inactive; otherwise, it is reestablished as soon as the fault is isolated at the corresponding hub ports. 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 prevent the communication through that hub for all nodes. Interlink faults do not prevent any node from communicating, as long as they do not affect all interlinks.

To tolerate a fault, a node that fails to communicate through a given hub as a consequence of that fault must continue to communicate through the other hub. A node that fails to communicate through a given hub will observe what Barranco et al. [2008] have called 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 or reception notified by the correct controller, that is, the one that has no problems to communicate (note that the fault model does not include so-called *byzantine* failures [Lamport, Shostak, and Pease, 1982]; thus it is assumed that there are no failure modes where a controller forges notifications, that is, generates notifications incorrectly; therefore, the controller that notifies is assumed to be the correct one).

If the controller that omits notifications is the non-transmission controller, the node does not even need to diagnose it as faulty—the other controller, the transmission controller, will be able to continue to receive and to transmit frames and the fact that the non-transmission controller omits notifications does not prevent the node from communicating. However, if the controller that omits is the transmission controller, the node must eventually diagnose it as faulty and discard it; otherwise, the node will not be able to transmit anymore. To overcome this problem, the node initiates a *transmission timer* (tx timer) when it requests a transmission: if the timer expires before the transmission controller notifies of a successful transmission, the node discards

the transmission controller and uses the other controller to transmit and receive.

Additionally, to prevent controllers from going into the *error-passive state*, in which they could inconsistently exchange frames (that is, in which some nodes could accept a given frame while other nodes reject that same frame), a CAN controller is discarded whenever its Transmission Error Counter (TEC) or its Reception Error Counter (REC) reaches a given threshold (that threshold is known in many CAN controllers as the *error warning limit*).

When the microcontroller discards a CAN controller for communication, it also has to check if that controller is the one which is currently marked as the transmission controller. If that is the case, it assigns the transmission controller role to the surviving controller.

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 next-to-last bit of a frame that have been identified for CAN (see Section 3.4.3). On the other hand, this inconsistent frame exchange may happen due to a stuck-at-recessive fault. Specifically, if the stuck-at-recessive fault prevents a controller from monitoring the traffic, or if it affects the controller's uplink and prevents the controller from aborting an ongoing frame that it rejects. As an example, consider a downlink that is stuck-at-recessive during the broadcast of a whole frame. In that case, the controller connected to that downlink will not receive the frame, whereas the other controllers will receive it—the frame has therefore been exchanged inconsistently due to a stuck-at-recessive. The ReCANcentrate media management takes into account the scenarios affecting the next-to-last bit of a frame to some extent, as well as the inconsistencies provoked by stuck-at-recessive faults, by assuming that a frame is not inconsistently exchanged more than a predefined number of consecutive times. This is explained in Chapter 6.

5.3.3. Driver architecture

Manuel Barranco designed the media management to be implemented as a software driver that abstracts away the details of the node architecture and the media replication. Figure 5.5 depicts the basic structure of the driver. It shows the peripherals the driver requires: two CAN controllers, *CAN 1* and *CAN 2* in Figure 5.5, and a timer to be used as the transmission timer (*tx timer*).

At the top part of the structure we can see the interface the driver provides to the application: the *driver interface*. 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 as if there was only a single CAN controller.

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 transmission controller, but it stores a copy of that frame in the driver's transmission buffer. The driver needs this copy for different management operations; for instance, if the driver diagnoses the transmission 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 driver's reception buffer, it is a buffer that accommodates the last frame received through ReCANcentrate. When the driver ac-

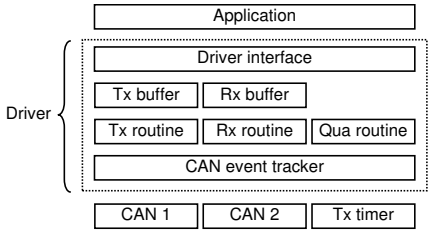

Figure 5.5.: Basic driver structure. (Reprinted from a paper we published [Barranco, Geßner, Proenza, and Almeida, 2010].)

cepts a frame reception, it immediately copies the frame from the hardware reception buffer of one of the controllers to the driver's reception buffer and releases the hardware reception buffers of both controllers. Afterwards, the application is informed of the reception of a frame.

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. An ISR, also known as an interrupt handler, is a routine that gets invoked in response to an interrupt. Usually a specific ISR is associated with a given interrupt by means of an *interrupt vector table* (IVT), which is a table in memory where each entry contains the memory address of an ISR and where each entry corresponds to a specific interrupt—the first table entry corresponds to interrupt one and contains the memory address of the ISR for interrupt one, the second table entry corresponds to interrupt two and contains the memory address of the ISR for interrupt two, and so forth.

The *tx routine* is executed when any of the two CAN controllers notifies of a transmission. The *rx routine* is executed when any of the two CAN controllers notifies of a reception. The *qua routine* is executed when the TEC or REC of any of the two CAN controllers reaches a specific threshold or when the transmission timer expires. To simplify the routines, Manuel Barranco considered that they cannot be nested, which requires that all of them have the same execution priority. These routines are explained in detail in Chapter 6.

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 a software 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 Chapter 6, the qua routine deactivates the controller and, thus, the rx routine would otherwise access the reception buffer of a deactivated controller.

The CAN event tracker functions as a dispatcher that decides which management routine must handle each notification. Moreover, it keeps track of which notifications have occurred and, thus, which routines it has triggered. This information is essential for the management routines

because they cooperate with each other (see Chapter 6) and must know which other routines have been triggered. Which routines have been triggered is indicated by several boolean variables, one for each type of notification, which are called *tracking variables*.

5.3.4. Hardware requirements of the driver

As can be deduced from the description of the driver architecture, in the previous section, the driver imposes some requirements on the hardware. These hardware requirements are the following:

- Availability of two CAN controllers. Two CAN controllers are needed so that a node can be connected to both ReCANcentrate hubs.
- Interrupt nesting with configurable interrupt priorities. Interrupt nesting refers to the preemption of a lower priority ISR by a higher priority ISR. This is needed to allow the CAN event tracker to preempt any management routine that is executing. Moreover, the driver must be able to assign a higher priority to the CAN event tracker than to the management routines so that the CAN event tracker can preempt the latter.
- Software interrupts. This is needed to allow the CAN event tracker to trigger the adequate media management routine when a notification occurs.
- Interrupt generation when a controller's transmission error counter (TEC) or receive error counter (REC) reaches a threshold, which must be below 128, the value at which a CAN controller enters the error-passive state. This is needed so that the CAN event tracker can be triggered once a CAN controller has detected too many errors, but before the controller enters the error-passive state.
- The CPU of the microcontroller must be fast enough to execute the CAN event tracker and any triggered tx routine or rx routine before the next *delivery event* takes place, that is, before the next frame is exchanged. The reason for this is that the tx and rx routines cooperate with each other and, thus, if the CPU were not fast enough, a tx or rx routine that is handling a previous delivery event could incorrectly cooperate with a tx or rx routine that is handling a subsequent delivery event.

5.4. Previous ReCANcentrate prototype

In order to verify the functionality and to measure the performance of ReCANcentrate—in particular of their hubs—Barranco et al. [2006a] implemented a prototype that had two ReCANcentrate hubs and three simplified nodes. In this section we describe the implementation of this previous prototype, as our prototype builds upon it. However, since both the hubs in the previous prototype and in our prototype were implemented using Field Programmable Gate Arrays (FPGAs), we begin this section with a brief introduction to FPGAs.

5.4.1. Brief introduction to FPGAs

FPGA stands for Field Programmable Gate Array, and it is described by the corresponding Wikipedia article [Wikipedia, 2010a] as follows:

> A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by the customer or designer after manufacturing—hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC) .... FPGAs can be used to implement any logical function that an ASIC could perform.
>
> FPGAs contain programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" .... Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.

FPGAs also have several I/O pins that can be used to input signals to the logic blocks or to output signals from the logic blocks. Figure 5.6 shows the internal structure of an FPGA.

The process of programming an FPGA can be summarized as follows [XESS Corporation, 2010]:

1. The logic circuit to be installed on the FPGA is designed using a hardware description language (HDL).

2. A *logic synthesizer* program transforms the HDL code into a so called *netlist*, which is a description of various logic gates and of how these logic gates are interconnected.

3. A *mapping tool* groups the logic gates of the netlist into several groups that can be fitted into the logic blocks of the FPGA.

4. A *place and route tool* assigns the groups of logic gates to specific logic blocks of the FPGA and specifies how the interconnections between the logic blocks are to be configured.

5. A software tool called a *bitstream generator* generates an appropriate bitstream from the specification that resulted from the previous step. This bitstream can then be downloaded into the physical FPGA (for instance, via a cable connecting the board that contains the FPGA to the parallel port of the computer where the bitstream is). After the download the FPGA performs the operations specified in the HDL code.

5.4.2. Hub implementation

Barranco et al. [2006a] implemented the coupler module and the fault-treatment module using a Xilinx Spartan-3 XC3S1000 FPGA embedded within an XSA-1000 board. Specifically, they

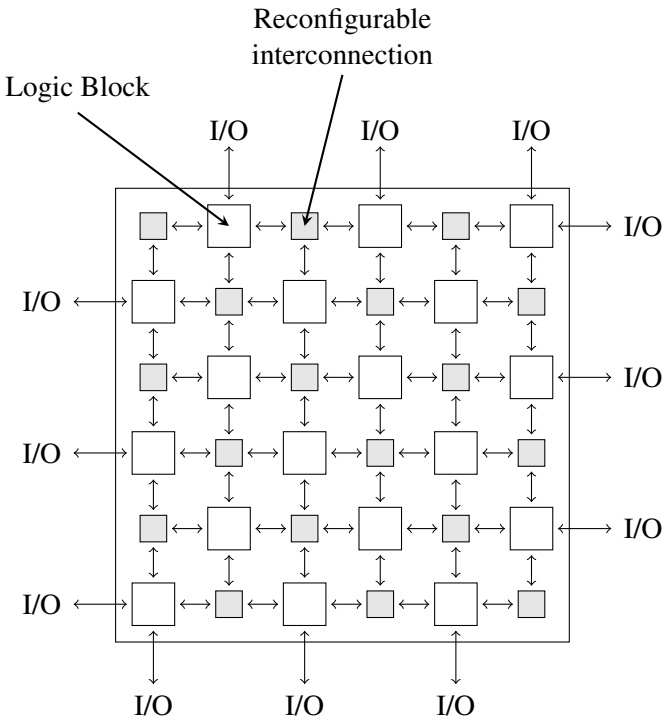

Figure 5.6.: Internal structure of a field-programmable gate array (FPGA). An FPGA is comprised of programmable logic blocks interconnected through reconfigurable interconnections and has several input/output (I/O) pins.

implemented the coupler module and the fault-treatment module using the VHSIC Hardware Description Language (VHDL), which is a particular HDL used to specify FPGA configurations. On the other hand, for the I/O module, they implemented "a dedicated board with four pairs of PCA82C250 CAN transceivers and four RJ45 plugs (one for each transceiver pair) providing the connection for four hub ports" [Barranco et al., 2006a]. Each of these hub ports could either be used as an interlink or as a link to a node, depending on the configuration specified in the VHDL file used to configure the FPGA [Barranco et al., 2006a]. Finally, "each link and each interlink were built using a dedicated UTP Cat 5 Ethernet cable, which contains 4 twisted pairs, allowing a fast implementation of both the uplink and downlink in a single cable, as well as both sublinks of an interlink" [Barranco et al., 2006a]. The electronic circuits they used for the I/O modules of their hubs are described in Section 5.4.4.

5.4.3. Node implementation

In their prototype Barranco et al. [2006a] did not implement the nodes as shown in Figure 5.4 (page 47), but with a single CAN controller instead, as shown in Figure 5.7. We call these nodes, which use a single CAN controller, *simplified nodes*. The approach used for the simplified nodes is similar to an approach proposed by Rufino et al. [1999]. As can be seen in Figure 5.7,

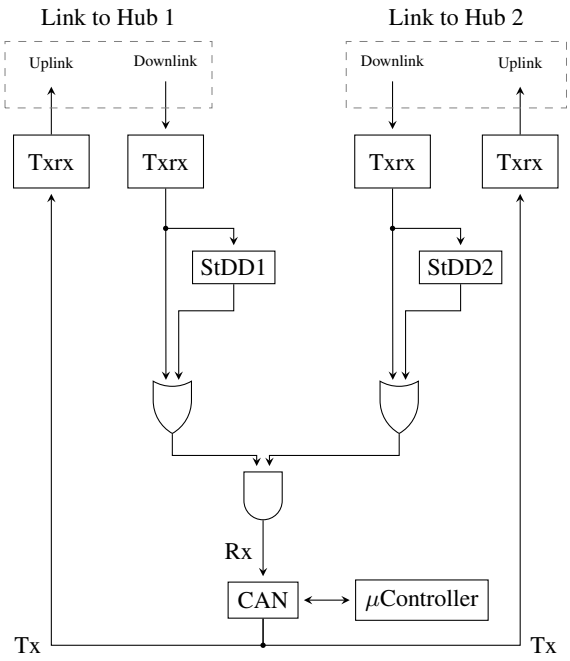

Figure 5.7.: ReCANcentrate node architecture using an approach inspired by Rufino et al. [1999]. This is the architecture that Barranco et al. [2006a] used for their simplified nodes.

it incorporates for each node one CAN controller (labeled CAN) with four CAN transceivers (labeled Txrx) grouped in two pairs. Each pair connects to a different hub using one transceiver for the downlink and the other one for the uplink. Each of the node's downlinks bifurcates into two branches after having entered the downlink's transceiver. One branch enters an OR gate while the other enters a *stuck-at-dominant detector* (StDD1 and StDD2 in Figure 5.7), whose output then enters the same OR gate.

Each of the stuck-at-dominant detectors is a simple circuit that outputs a logical 0 as long as the number of consecutive dominant bits received through the corresponding downlink does not exceed a specific threshold; otherwise, when the threshold is exceeded, the circuit permanently outputs a logical 1.

In the absence of faults, each of the OR gates receives a stream of zeros from its attached stuck-at-dominant detector. This means that each of the OR gates has a permanent zero at one of its inputs. The output of the OR gates will therefore be what comes through the other input, the one without the permanent zero. The inputs without the permanent zero are the downlinks, thus, each OR gate outputs the stream received through the corresponding downlink. Moreover, as ReCANcentrate enforces a single broadcast domain, each bit value is quasi-simultaneously received on both of the node's downlinks. Therefore, during any given bit time, both OR gates output the same bit value. This means that the AND gate receives the same value on both its inputs and, thus, outputs that bit value, which then enters the CAN controller. In summary, in the absence of faults, the node's CAN controller receives the bit stream coming from both

downlinks, which is the bit stream being broadcast by both hubs.

If a fault manifests as a stuck-at-recessive downlink, the sequence of 1's coming from that downlink enters the AND gate. Each bit coming from the non-faulty downlink will then be ANDed together with the 1's from the stuck-at-recessive downlink and, therefore, the stream of bits that gets to the CAN controller is the one from the non-faulty downlink.

If a fault manifests as a stuck-at-dominant downlink, the corresponding stuck-at-dominant detector is the one producing the sequence of 1's that isolates the faulty contribution received through that downlink.

Barranco et al. implemented the CAN nodes using COTS components only: the single CAN controller and the CPU of each node were part of a PIC microcontroller [Microchip, 2004]; the stuck-at-dominant detectors were implemented using a simple electronic circuit comprised of a capacitor, a resistor, and a NOT gate; and the transceivers were the PCA82C250 CAN transceivers from Philips Semiconductors [1997].

Unfortunately we cannot recommend the simplified nodes for highly fault-tolerant systems. First, they do not tolerate faults that manifest as a bit-flipping downlink. Second, there is another big disadvantage to this approach, and in general to any approach where a node uses a CAN controller that sends the same signal through both uplinks. Notice in Figure 5.7 that since the CAN controller sends error flags through both uplinks in response to errors it detects in any of the downlinks, both hubs observe these error flags at the respective uplink ports. This will likely cause both hubs to isolate the node even when the fault only affects the connection of the node to one of the hubs. Finally, this approach would require higher level protocol layers to tolerate the data inconsistency scenarios described in Section 3.4.3.

5.4.4. Electronic circuits

This section contains two schematic diagrams of electronic circuits that Barranco et al. [2006a] developed and implemented for their prototype. We implemented them again in our prototype as will be explained in Chapter 7.

Electronic circuit for an I/O module port of a ReCANcentrate hub or node

Figure 5.8 shows the schematic of the electronic circuit that Barranco et al. [2006a] designed to implement the port of an Input/Output module of a hub or of a node. If the circuit is used for a hub, then we can plug into such a port (into the RJ45 connector) either a link from a node—comprised of an up- and a downlink—or an interlink from the other hub—comprised of an incoming and an outgoing sublink. If the circuit is used for a node, we can plug into such a port a link to a hub—comprised of an up- and a downlink.

The transceivers used are PCA82C250 CAN transceivers [Philips Semiconductors, 1997]. In the schematic, the transceiver shown in the top is the *receiving* transceiver; whereas the transceiver shown in the bottom is the *transmitting* transceiver. These transceivers have each a pin for the CANH and the CANL wire and are labeled accordingly in the schematic. Moreover, these transceivers have the following additional pins:

Slope resistor input pin (R_S): used to switch between high-speed, slope control, and standby

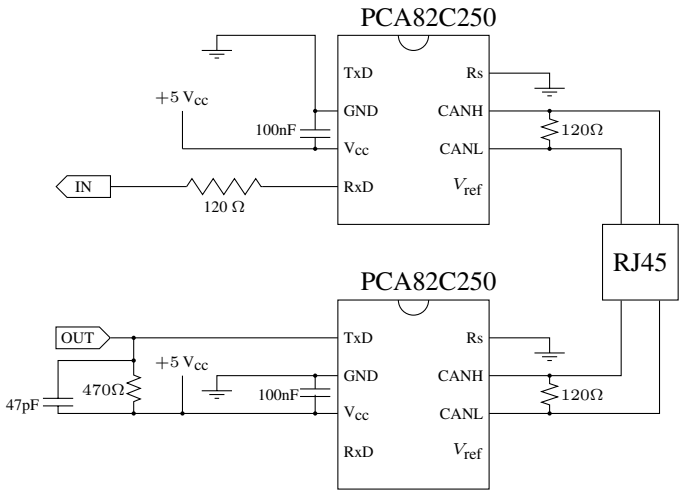

Figure 5.8.: Electronic circuit for a port of the I/O module of a hub or node.

mode [Philips Semiconductors, 1997]. It is connected to ground in the schematic, which selects the high-speed mode.

Reference voltage output pin (V_{ref}): provides a reference voltage. It is left unconnected in the schematic.

Supply voltage pin (V_{CC}): it is connected to 5 V and protected from noise by a 100 nF decoupling capacitor, that is, a capacitor that removes momentary glitches in the power source.

Ground pin: connected to ground.

Transmit data input pin (TxD): connected to the source of the logic level signals which are to be converted to a differential voltage between the CANH and CANL pins. It is left unconnected in the receiving transceiver. In the transmitting transceiver it is attached to a resistor-capacitor circuit, which is used in case the wire labeled OUT is connected to a 3.3 V output pin that needs to be pulled up to 5 V (that would be the case if the CAN controller or the hub to which the OUT wire is connected operates at 3.3 V instead of the transceiver's 5 V).

Receive data output pin (RxD): connected to the destination of the logic level signals corresponding to the differential voltage between the CANH and CANL pins. It is left unconnected in the transmitting transceiver.

When the circuit of Figure 5.8 is used to implement an input port of a hub, the wire labeled IN connects the RxD pin of the receiving transceiver to the pin of the coupler module corresponding to that port contribution: B_i (of Figure 5.3, page 45) when the contribution is from a node or B'_{0i} when the contribution is from a hub. When the circuit is used to implement an output port of a hub, the wire labeled OUT connects the TxD pin of the transmitting transceiver to the coupler

module pin that outputs the hub's contribution (B_0) or to the pin that outputs the result of the coupling of both hubs' contributions (B_T).

On the other hand, when the circuit is used by a simplified node, the IN wire is the wire that splits in two and connects both to an OR gate and to a stuck-at-dominant detector (see Figure 5.7), and the OUT wire is the wire that carries the output of the simplified node's CAN controller.

Regardless of whether the circuit is used to implement the port of a hub or a node, the port is implemented with an RJ45 plug, and the link or interlink is implemented with a UTP Cat 5 Ethernet cable, of which only four wires are used: two for the CANH and CANL of the incoming signal, and two for the CANH and CANL of the outgoing signal. Finally, to avoid signal reflections, both CANH and CANL wire pairs are terminated using a 120 Ω terminating resistor.

Electronic circuit to connect an external oscillator

In order to properly communicate, the oscillators of the ReCANcentrate hubs and the ReCANcentrate nodes must all have frequencies that are multiples of each other. The frequencies of these oscillators can then be scaled by each node and hub so that they all use a clock with the exact same frequency to sample each other's transmitted bits. If they did not use the same clock frequency for sampling, receiving nodes and hubs would not be able to synchronize themselves with the leading transmitter. Refer back to Section 3.3, for a more detailed discussion on the synchronization between nodes.

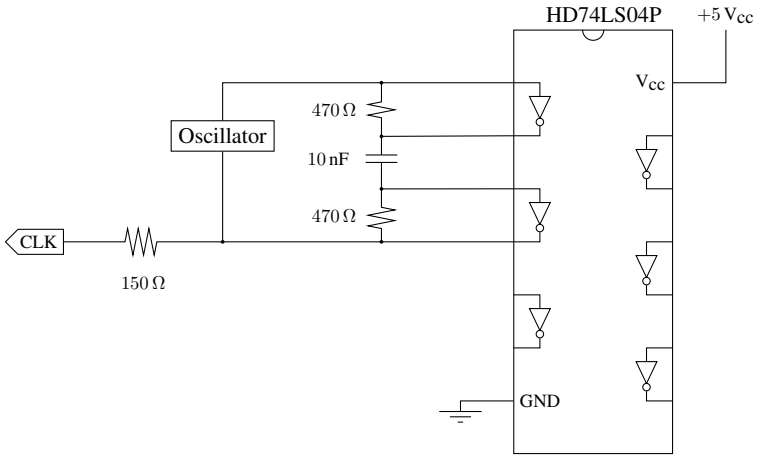

Figure 5.9.: Electronic circuit to attach an external oscillator.

Because the internal oscillators of the FPGAs, where the hubs were synthesized, were not a multiple of the internal oscillators of the simplified nodes, Barranco et al. had to attach an external oscillator to either the hubs or the nodes. Specifically, either an oscillator had to be attached to the hubs to match the frequency of the nodes' oscillators or, alternatively, an oscillator had to be attached to the nodes to match the frequency of the hubs' oscillators. This is what the electronic circuit shown in the schematic of Figure 5.9 is for.

Microcontrollers and FPGAs usually have a dedicated pin to which an external clock source can be connected to. By inserting an appropriate oscillator into the electronic circuit, by then connecting the wire labeled CLK to that pin, and by configuring the microcontroller or FPGA to use an external clock source, we can ensure that the nodes and the hubs have frequencies multiples of each other.

Part II.

Project specific tasks

6. Final design of the media management driver for the ReCANcentrate nodes

This chapter describes the final design of the media management driver that allows the ReCANcentrate nodes to communicate through the replicated media provided by the hubs while tolerating both the crash of a CAN controller and the occurrence of faults that manifest as stuck-at or bit-flipping bits in the channel. The design is based on the original design by Manuel Barranco (the flowcharts of which are printed in Appendix A), but corrects and improves a few things. We do not discuss the original design and we do not compare it to the final design; instead, we only discuss the final design, which is what we implemented for the nodes of our prototype.

Note that Manuel Barranco did not only create the initial design of the media management driver, but that he also provided a fair amount of input to the final design.

Remember from Section 5.3.3 that the driver for the nodes has three main components: the CAN event tracker, the media management routines, and the driver interface. This chapter begins by first describing in detail each of the media management routines: the tx routine, the rx routine, and the qua routine. Afterwards, it introduces the *transmission request routine* (*tx request routine*), which is part of the driver interface. Then it describes three examples that illustrate how the media management routines work and how they interact with the CAN event tracker and the tx request routine. Finally, the chapter describes the fault-tolerance capacities of the media management driver, focusing on how the CAN inconsistency scenarios from Section 3.4.3 are tolerated.

6.1. Media management routines

As described in Section 5.3, when a delivery event occurs at a ReCANcentrate node, it is expected that each of the node's CAN controllers notifies about the transmission or reception that has occurred. This notification is done through interrupts which invoke the CAN event tracker. The CAN event tracker invocation then checks why it was invoked and then subsequently triggers the tx routine, if it was invoked because of a transmission; or the rx routine, if it was invoked because of a reception.

In the absence of faults, there will be two CAN event tracker invocations for each delivery event—one for each controller. If a transmission occurred, one invocation will trigger the tx routine and the other invocation will trigger the rx routine; if a reception occurred, both invocations will trigger an rx routine. In any case, two media management routine executions will be triggered and one of the executions will execute before the other. In that case, the two executions must collaborate to handle the delivery event.

On the other hand, in the presence of faults, one of the CAN controllers may omit a notification and only one routine execution may be triggered because of the delivery event. In that case, the

single routine execution must handle the event alone.

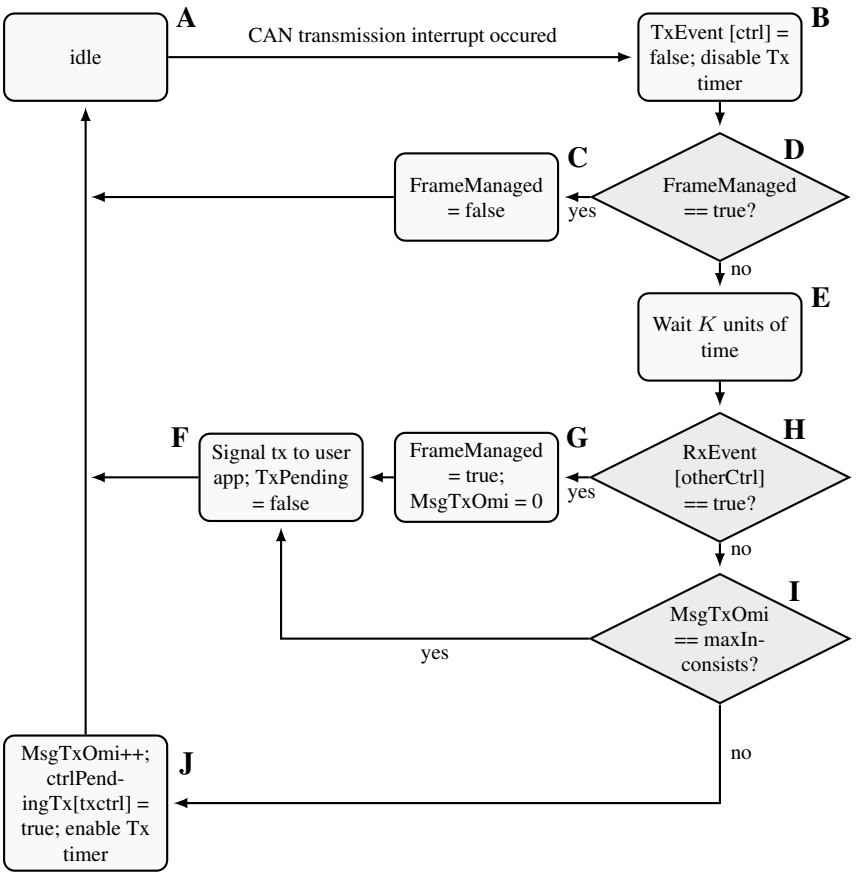

Figure 6.1.: Final design of the *tx routine*.

6.1.1. The tx routine

The flowchart for the tx routine is shown in Figure 6.1. The routine starts with the reset of both a tracking variable and the transmission timer (this is shown in block B of Figure 6.1; `ctrl` refers to the controller that notified the transmission). The tracking variable was set by the CAN event tracker and indicates that the tx routine has been triggered because of a notification by the controller `ctrl`. The transmission timer was first introduced in Section 5.3.2. As explained in that section, it counts the time that the transmission controller has available for the transmission of a frame before the driver considers the transmission controller to be faulty. Next, since a frame transmitted by the transmission controller should be received by the non-transmission controller, the tx routine checks if the rx routine was triggered first and if it has verified that the non-transmission controller has received the frame that the transmission controller had transmitted. For this purpose it consults the driver variable `FrameManaged` (decision D in Figure 6.1). 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 (block C). If `FrameManaged` is false, the routine waits K units of time to give the non-transmission controller enough time to notify the reception of the transmitted frame (block E)—the non-transmission controller may have been slower to notify so the wait may be necessary. Afterwards, it consults the tracking variable `RxEvent[otherCtrl]` to check if this notification has occurred (block H). If it has, the routine sets `FrameManaged` to `true` and resets the driver variable `MsgTxOmi`, whose role will be explained later, to zero (block G). Setting `FrameManaged` to true (in block G) informs the rx routine which is going to be executed next that the tx routine has already verified that the non-transmission controller received the transmitted frame; the rx routine will therefore not have to verify it again. Finally, (in block F) the tx routine indicates to the application that the frame has been successfully transmitted and resets the driver variable `TxPending`, 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* (see Section 5.3.2). When this happens, it checks if the number of omissions detected so far is equal to the maximum number of consecutive inconsistencies (decision I). If so, the tx routine assumes that this inconsistency is not due to one of the data inconsistency scenarios discussed in Section 3.4.3, but due to a permanent stuck-at-recessive fault that prevents the non-transmission controller from communicating; thus, it simply indicates to the application that the frame has been successfully transmitted (block F). Otherwise, it increases the omission counter and requests a retransmission of the frame through the transmission controller (block J)—note that the retransmission may cause the reception of duplicated frames at some nodes.

6.1.2. The rx routine

The flowchart for the rx routine is shown in Figure 6.2. Just like the tx routine, the rx routine starts by resetting the tracking variable that indicates that it has been triggered because of a notification by the corresponding controller (block B)—although this time the notification was a reception event instead of a transmission event. The next thing that the rx routine does is to check the driver variable `FrameManaged` (decision E). If the variable is true, it means that the frame whose reception has launched the rx routine has either already been managed by the tx routine or that it has already been managed by another execution of the rx routine which was triggered by the other controller; in any case, if the `FrameManaged` variable is true, all the rx routine must do is reset the variable (block D) and release the reception buffer of its corresponding controller (block C).

If the `FrameManaged` variable is false (in block E), the rx routine knows that it is the one that is executing first; it therefore waits K units of time to give the other controller time to notify (block F). Then, the rx routine checks whether or not the other controller has triggered a tx routine (decision I). If it has, this means that the frame the rx routine is managing is actually a frame that was transmitted by the other controller. In that case, the rx routine sets `FrameManaged` to `true` to inform the tx routine—which will be executed next—that it already managed the transmission and resets the driver variable `MsgTxOmi` (block H). Next, the rx routine indicates to the application that the frame has been successfully transmitted and resets the driver variable

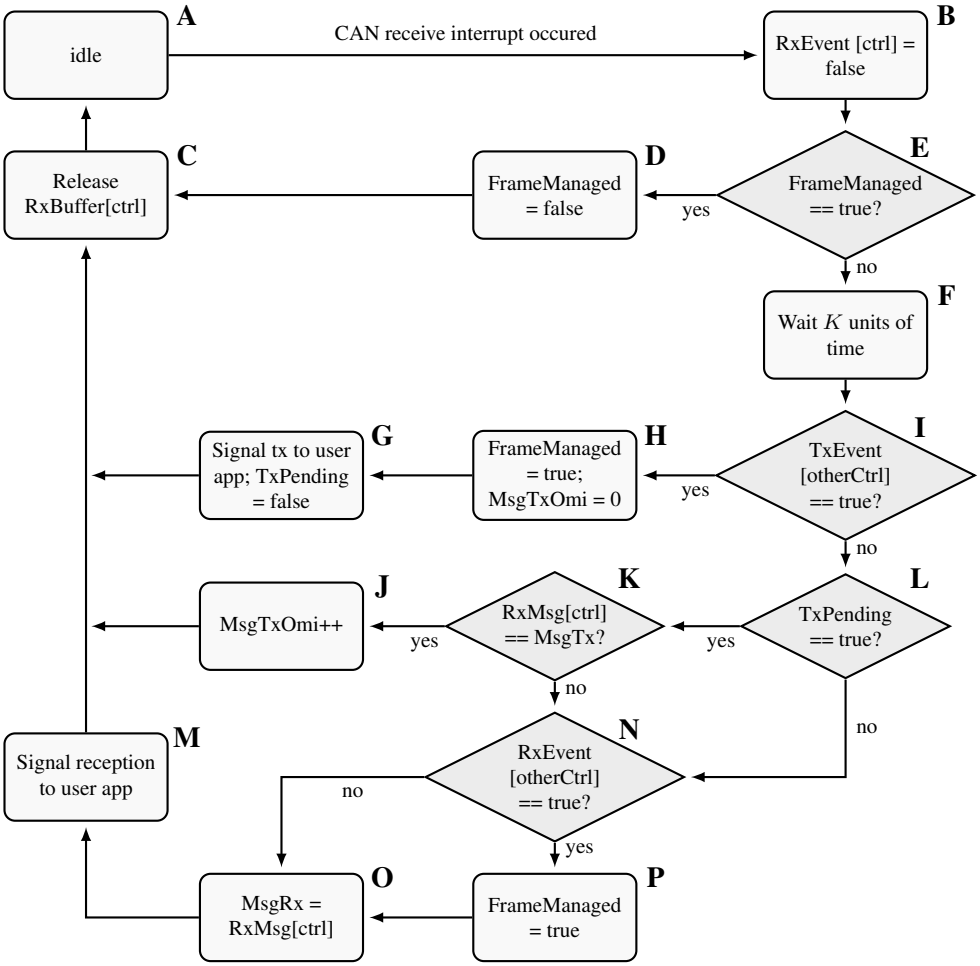

Figure 6.2.: Final design of the *rx routine*.

`TxPending` (block G), thus indicating that the transmission of the frame is no longer pending. Finally, the rx routine releases the reception buffer of the controller whose notification caused the rx routine to be executed (block C).

In case the rx routine detects (in decision I) that the other controller has not triggered a tx routine, the rx routine still needs to check if it is managing a frame transmitted by the other controller. The reason is that the other controller could omit the notification of a transmission due to a fault—it could crash before triggering the tx routine—or it could omit the notification of the transmission due to a CAN inconsistency scenario which led the other controller 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 `TxPending` is true (decision L) and (in decision K) if the frame placed at the tx buffer of the driver (`MsgTx`) is equal to the frame contained within the hardware reception buffer of the corresponding controller (`RxMsg[ctrl]`). If so, the rx routine assumes that an omission discrepancy occurred and, thus,

increases the omission counter (`MsgTxOmi`) (block J). Note that in case the omission is due to a permanent fault, that is, the transmission controller has crashed, then the transmission timer will eventually expire and the qua routine will carry out the actions needed to tolerate the fault (see Section 6.1.3). If the node had a frame pending for transmission (decision L, branch labeled *yes*), but the controller did not receive the pending frame (decision K, branch labeled *no*), then the controller received a frame from another node. The controller has also received a frame from another node if there was no transmission pending on the node that is executing the rx routine (decision L, branch labeled *no*).

Either way, when the rx routine determines that the frame came from another node, it checks if the other controller has also received it (decision N). If it has, the other controller will have triggered another execution of the rx routine. Therefore the current rx routine execution sets the driver variable `FrameManaged` to `true` (in block P) to indicate to the other rx routine execution that it has already managed the reception of the frame; additionally, it copies the received frame from the controller's hardware reception buffer to the driver's reception buffer (block O). If the other controller has not received the frame, the rx routine proceeds directly to copying the received frame to the driver's reception buffer (block O again, but now through the branch labeled *no* outgoing from decision N). In any case, at the end, the rx routine sets a flag of the driver to signal to the user application that the frame was received (block M) and releases its controller's hardware reception buffer (block C).

6.1.3. The qua routine

When the transmission error counter (TEC) or the reception error counter (REC) of a controller reaches a given threshold (the error warning limit), this causes an interrupt that invokes the CAN event tracker. The CAN event tracker then determines that it was called because of an error warning and invokes the qua routine, which then quarantines (deactivates) the controller whose TEC or REC has reached the error warning limit. Similarly, when the transmission timer expires, the qua routine is also invoked through the CAN event tracker in order to quarantine the transmission controller as the expiration of the transmission timer implies that the transmission controller has crashed.

The flowchart for the qua routine is shown in Figure 6.3. The routine starts by checking if the controller that is to be quarantined is active (decision B in Figure 6.3). If it is not active, it has already been quarantined by a previous execution of the qua routine (the qua routine could be invoked twice for the same controller if the error warning limit is reached and shortly after that the transmission timer expires). So, if the controller has already been quarantined, then there is nothing left to do for the qua routine and it simply finishes. On the other hand, if the controller is still active, then the qua routine is marked as deactivated and then reset (this is done in block C). Next, the qua routine must check whether the deactivated controller was the transmission controller (decision G). If it was not the transmission controller, the qua routine has finished. But if it was, then the qua routine has to perform a set of actions needed to start using the surviving controller as the new transmission controller. These actions are the following. First, the routine disables the transmission timer (block F): this prevents the timer from unnecessarily expiring in case the qua routine was triggered because of an error warning and the timer was still running. Afterwards, the qua routine checks if the other controller is non-faulty, that is, if

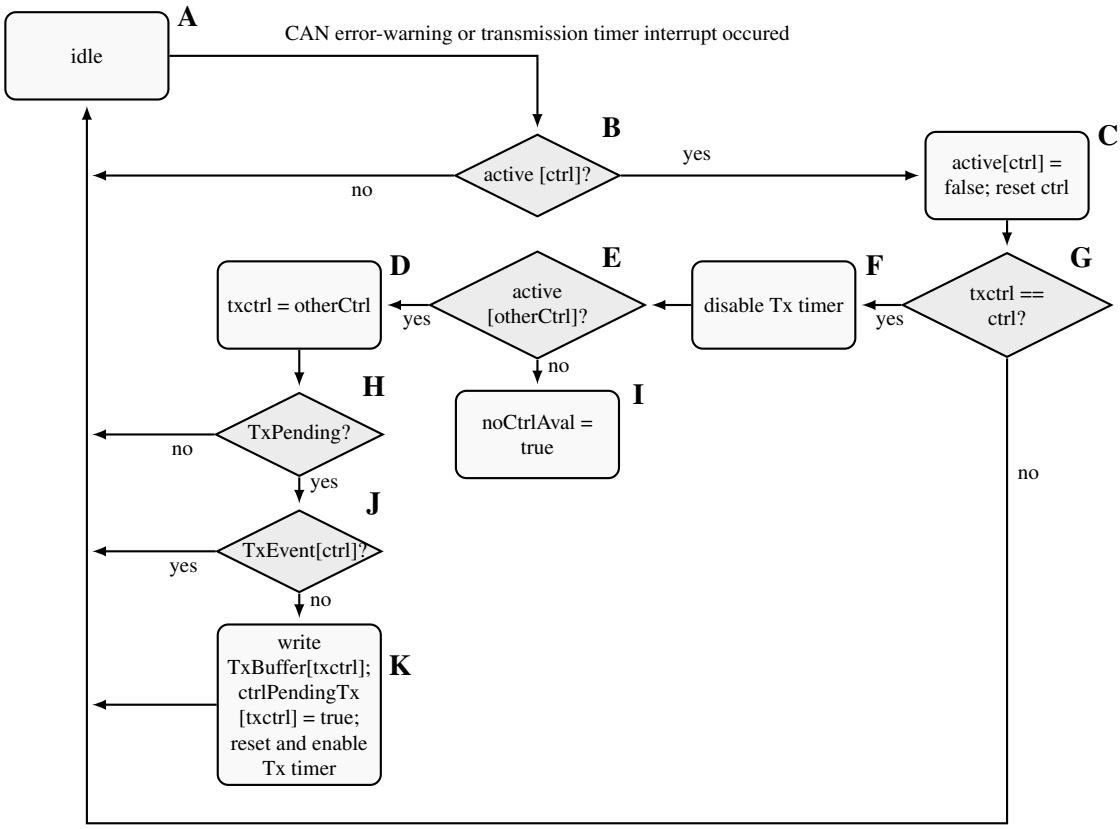

Figure 6.3.: Final design of the *qua routine*.

the other controller is *active* (decision E). If not, this means that there are no controllers left for the node to communicate; thus, all the routine can do is inform the application that there are no controllers available anymore (block I). If the other controller is still active (decision E, branch labeled *yes*), the routine marks the other controller as the new transmission controller (block D) and, then, checks whether or not a frame's transmission is pending (decision H). If it is pending, it could be necessary to request the transmission of the frame through the new transmission controller. To decide if it has to transmit the frame through the new transmission controller, the qua routine checks the tracker variable that indicates whether the controller that was deactivated previously (in block C) notified the transmission (decision J). If it is true, then the controller that was deactivated previously (in block C) was able to finish the transmission before the qua routine was invoked; thus, the qua routine must not instruct the transmission through the new transmission controller. However, if the variable is false, then the deactivated controller was not able to finish the pending transmission before the qua routine was invoked and a transmission through the new transmission controller must be instructed (block K).

6.2. The tx request routine

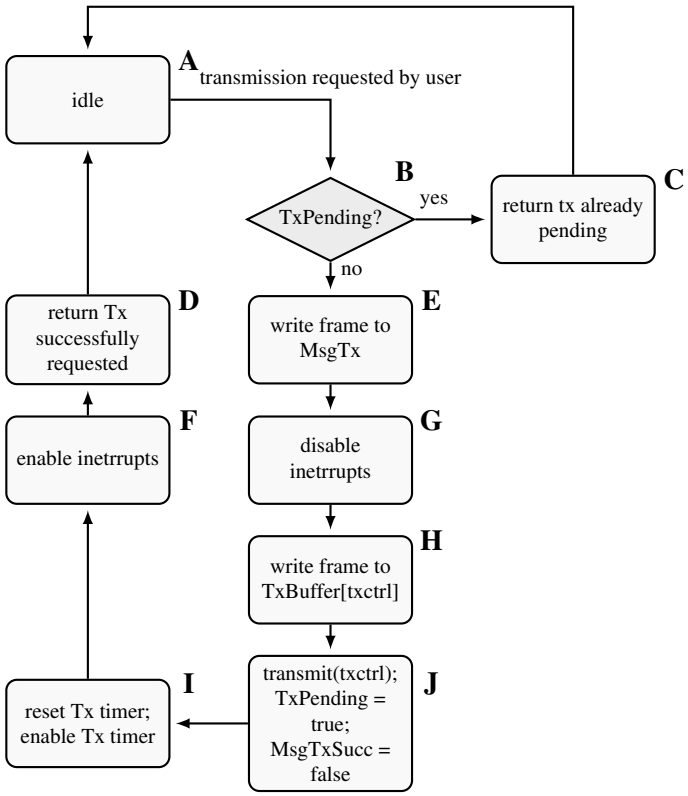

Figure 6.4.: Final design of the *tx request* routine.

In contrast to the media management routines, the tx request routine—which is part of the driver interface (see Figure 5.5 on page 50)—is directly called by the user application running on the node instead of being called through software interrupts generated by the CAN event tracker. The user application calls this routine whenever it wants to transmit a frame through the replicated channel provided by the ReCANcentrate hubs. When the routine is called, as the flowchart in Figure 6.4 shows, it first checks whether a transmission is already pending (decision B). If it is, this is simply communicated to the user application (block C) and nothing else is done in the routine. If no transmission is pending yet, the routine writes to the driver's transmission buffer the frame whose transmission is requested (block E), which is passed as a parameter to the tx request routine. Afterwards, the routine temporarily disables the interrupts to the CPU (block G). This is done in order for the routine to have exclusive access to the controllers, the transmission timer, and the driver's variables. If the interrupts were not disabled, one of the media management routines could preempt the CPU while the tx request routine is being executed. The preempting media management routine could then, for instance, modify the transmission buffer of the transmission controller or change any of the driver's state variables (such as the variable that indicates whether a transmission is pending or the variable that indicates which is

the transmission controller). This would have unintended consequences.

After the interrupts have been disabled, the tx request routine copies the frame to be transmitted to a second buffer: the hardware transmission buffer of the transmission controller (see block H). Then, the routine instructs the transmission controller to transmit the frame and it updates the driver's status variables to indicate that the transmission is pending and that the transmission has not (yet) been carried out successfully (block J). Note that when the routine instructs the transmission controller to transmit the frame, the frame is not transmitted immediately; the transmission controller will first have to gain access to the medium: it will have to wait until the communication channel is idle and then it will have to win the arbitration if there are other nodes that also attempt to transmit a frame. Anyway, after the transmission through the transmission controller has been instructed and the status variables have been updated, the routine needs to set up the transmission timer (block I). It resets the transmission timer so that it starts counting again from zero up to the transmission timeout and it then enables the timer. Once this is done, the interrupts can be enabled again (block F) and the user application can be informed that the transmission was successfully requested (block D).

6.3. Example executions

To better illustrate how the CAN event tracker and the media management routines interact with each other, and to further clarify how the media management routines work, this section describes three example executions of the CAN event tracker and media management routines.

6.3.1. Fault-free reception

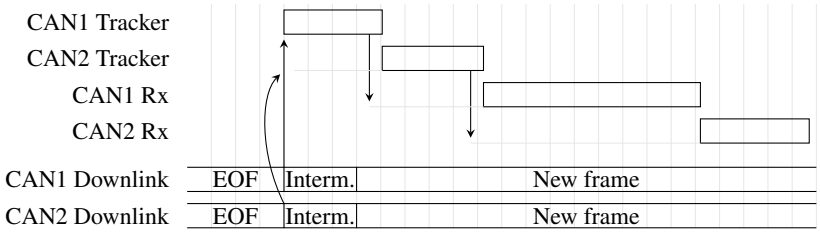

Figure 6.5.: Example execution of the reception of a data frame.

In this section we describe what occurs at a node when a frame is received from another node and there are no faults. This scenario is illustrated in the chronogram of Figure 6.5. CAN1 and CAN2 are the CAN controllers of the node. The bottom of the figure shows what each of the two CAN controllers receives through its downlinks, which is the same for both of them. The chronogram starts with the last few bits of the *end of frame* field (EOF) transmitted by another node. When the last EOF bit has been received by the controllers, both controllers

notify the reception of the transmitted frame. They do this through interrupts which invoke the CAN event tracker (shown as upward arrows in the figure). More precisely, each of the two interrupts invokes the CAN event tracker ISR corresponding to the controller that generated these interrupts.

The first CAN event tracker ISR to execute is the one for the CAN1 controller. It determines that it was invoked because of a reception at the CAN1 controller and, thus, indicates that fact in a corresponding tracking variable—it sets `RxEvent[CAN1] = true`—and triggers the rx routine for the CAN1 controller by generating an appropriate software interrupt (represented as a downward arrow). When it finishes, the pending CAN2 tracker ISR executes, which has a higher priority than the now also pending ISR of the CAN1 rx routine. The CAN2 tracker ISR also determines that it was invoked because of a reception and, thus, indicates in a corresponding tracking variable that a reception at the CAN2 controller occurred—`RxEvent[CAN2] = true`—and triggers the rx routine for the CAN2 controller. Now the ISRs of the CAN1 rx routine and the CAN2 rx routine are pending.

In the scenario shown in the figure the CAN1 rx routine ISR executes before the CAN2 rx routine ISR. Referring back to the flowchart of the rx routine (Figure 6.2), the ISR of the CAN1 rx routine resets to false the tracking variable that the CAN1 tracker ISR had just set to true (block B in Figure 6.2) and then determines that it has to manage the frame just received at the CAN1 controller (decision E). It waits K units of time for the CAN2 controller to notify the delivery event (block F), which in the current scenario is unnecessary because the CAN2 controller already notified the reception of a frame—but in general there could be a delay between the notifications of the two CAN controllers, making the wait necessary. After the wait, the ISR for the CAN1 rx routine determines that the tracking variable that indicates that the CAN2 controller notified of a transmission is not set to true (decision I). It then further checks whether there is a transmission pending (decision L). Let us assume that there is not, that is, that there was no recent call to the tx request routine. The ISR for the CAN1 rx routine then moves on to check whether the tracking variable that indicates that the CAN2 controller notified a reception is set to true (decision N). It is: the CAN2 tracker ISR had set it to true just before the CAN1 rx routine ISR started its execution; thus, the CAN1 rx routine ISR knows that the CAN2 rx routine ISR is going to execute. So, to tell the CAN2 rx routine ISR that the reception of the frame has already been managed, the CAN1 rx routine ISR sets the `FrameManaged` variable to true (block P). Then, all that is left to do is to copy the received frame from the CAN1 controller's hardware reception buffer to the driver reception buffer (block O), signal to the user application that a frame was received (block M), and release the hardware reception buffer of the CAN1 controller (block C).

Once the ISR for the CAN1 rx routine finishes, the ISR for the CAN2 rx routine starts its execution (Figure 6.5). Referring again to the rx routine flowchart (Figure 6.2), the ISR for the CAN2 rx routine resets to false the tracking variable that indicates that the CAN2 controller notified a reception (block B)—the variable was set to true by the CAN2 tracker ISR that executed previously. Then it checks if it has to manage the reception of the frame by checking the value of the `FrameManaged` variable (decision E). As the CAN1 rx routine ISR has previously set that variable to true, the CAN2 rx routine ISR knows that it must not manage the delivery event again. So it merely resets that variable to false (block D) and releases the hardware reception buffer of the CAN2 controller (block C).

6.3.2. Fault-free transmission

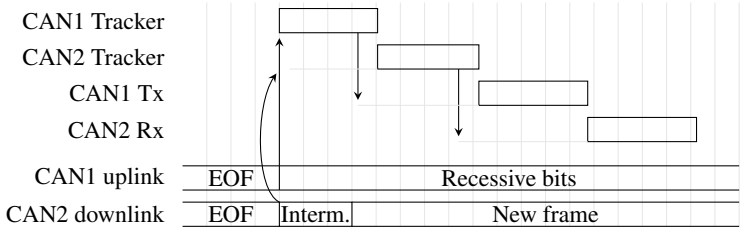

Figure 6.6.: Example execution of the transmission of a frame.

The next example execution we consider is a fault free transmission. Figure 6.6 shows the corresponding chronogram. As opposed to the previous example scenario, this time we have to differentiate between the roles of the two CAN controllers: CAN1 is the transmission controller and CAN2 is the non-transmission controller. The bottom of the chronogram shows for the CAN1 controller what it transmits through its uplink, and it shows for the CAN2 controller what it receives through its downlink. The *end of frame* field (EOF) shown at the beginning of the chronogram is being transmitted by the CAN1 controller and received by the CAN2 controller. After the EOF, the CAN1 controller does not transmit any other frame; and the new frame that is depicted at the part of the chronogram corresponding to the CAN2 downlink, after the intermission, is a frame coming from another node.

As in the previous example, the two CAN controllers notify the delivery event after the last EOF bit. However, what the CAN1 controller notifies now is a transmission, while the CAN2 controller continues to notify a reception. Again, both notifications are interrupts that invoke the corresponding CAN event tracker ISR.

Of the two tracker ISRs the first to execute is the one for the CAN1 controller. As always, it checks why it was invoked and, this time, it determines that it was invoked because of a transmission notification from the CAN1 controller. Thus, it keeps track of that fact by setting the appropriate tracking variable, `TxEvent[CAN1] = true`, and triggers the ISR for the CAN1 tx routine by generating an appropriate software interrupt. This makes the CAN1 tx routine ISR pending to be executed.

Next the tracker ISR for the CAN2 controller executes because it was pending and has a higher priority than the now also pending CAN1 tx routine ISR. It determines that it was invoked because of a reception notification by the CAN2 controller; keeps track of that fact, that is, it executes `RxEvent[CAN2] = true`; and generates a software interrupt for the CAN2 rx routine ISR.

Now the ISR for the CAN1 tx routine executes—it has the same priority as the CAN2 rx routine ISR, so, another valid scenario would be one where the CAN2 rx routine ISR executes before the CAN1 tx routine ISR. To describe what the CAN1 tx routine ISR does we will refer

back to the flowchart of Figure 6.1. First, it resets the tracking variable that the CAN1 tracker ISR had recently set and disables the transmission timer (block B). Then it checks whether it has to manage the frame or whether it has already been managed (decision D). As it has not been managed yet, the ISR continues by waiting K units of time for a notification of the CAN2 controller (block E)—the notification has already occurred in the scenario we are describing, but it might not have occurred yet in another scenario where there is a significant delay between the notifications of the two CAN controllers. After the wait, the ISR checks whether the corresponding tracking variable indicates that the CAN2 controller notified a reception (decision H). As it has, this means that the CAN2 controller has received the frame that the CAN1 controller transmitted. Thus, the tx routine ISR flags the frame as already been managed—it sets `FrameManaged = true`—and resets the omission counter to zero (block G). Finally, the ISR signals a successful transmission to the user application and marks the transmission as no longer pending (block F).

Once the ISR for the CAN1 tx routine has finished its execution, the ISR for the CAN2 rx routine starts. Referring to the flowchart of the rx routine in Figure 6.2, the ISR simply resets the tracking variable that the CAN2 tracker ISR had set to true (block B), it determines that the frame has already been managed (decision E), it resets to false the status variable that indicates that the frame has been managed (`FrameManaged`, block D), and releases the hardware reception buffer of the CAN2 controller (block C).

6.3.3. Example involving all four routines

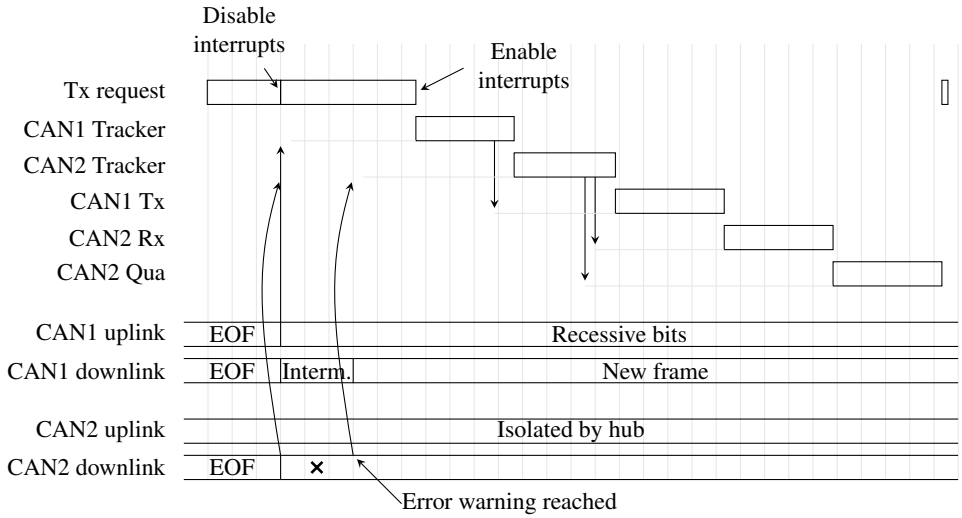

Figure 6.7.: Example execution involving a tx request, a rx routine, a tx routine, and a qua routine.

The final example we are going to describe involves all four media management routines. Its chronogram is shown in Figure 6.7. CAN1 is the transmission controller and CAN2 is the non-transmission controller.

The chronogram starts with the last few bits of the *end of frame* field (EOF) that the CAN1 controller transmits. These bits are received by the CAN1 and CAN2 controllers through their downlinks. Moreover, the chronogram shows that the uplink of the CAN2 controller has been isolated by the corresponding hub due to previous errors (which are not shown in the figure). When the frame transmitted by the CAN1 controller ends, CAN1 notifies a transmission and CAN2 notifies a reception. However, the CAN event tracker ISRs cannot start to execute because interrupts have been disabled by the tx request routine, which was called by the user application just before the CAN1 controller finished its transmission. The ISRs of the CAN event tracker are therefore postponed until the tx request routine enables interrupts again. Additionally, before interrupts are enabled again, the CAN2 controller detects an erroneous dominant bit during the second bit of the intermission. The CAN2 controller will mistake this for the start of an overload flag and attempt to transmit its own overload flag, which fails because its uplink is isolated (CAN2 will not monitor at its downlink the dominant bits that constitute the overload flag). Specifically, it detects a bit error during the third intermission bit. In this example, it is assumed that before the last frame was received at the CAN2 controller, its transmission or reception error counter had already nearly reached the error warning limit; thus, the error it detects makes one of its error counters cross the error warning limit. This causes an additional interrupt to the CAN2 event tracker ISR.

When the interrupts are enabled again, the CAN1 event tracker ISR preempts the tx request routine and starts to execute. It then determines that it was invoked because of a transmission by the CAN1 controller and therefore triggers the CAN1 tx routine ISR through a software interrupt. Next, the CAN2 event tracker ISR executes. It determines that it was invoked both because of a reception on the CAN2 controller and because of an error warning that occurred at the CAN2 controller. It therefore triggers both the CAN2 rx routine ISR and the CAN2 qua routine ISR and then finishes. At this point there are three media management routines pending to start their execution. As explained in Section 5.3.3, the implementation must ensure that the qua routine is executed after a rx routine, but apart from this requirement, the pending media management routines could execute in any order.

In the chronogram the next routine to execute is the CAN1 tx routine ISR. Its execution trace is exactly the same as in the previous example, so we do not explain it again. When it finishes, the CAN2 rx routine ISR starts its execution. Its execution trace is also the same as in the previous example, so we do not explain it again either.

Next, the pending CAN2 qua routine ISR executes. To explain its execution trace we refer to the flowchart of Figure 6.3. First, it checks whether the faulty controller that is to be deactivated, CAN2, is marked as active or not active (decision B). As there was no previous execution of a qua routine for the CAN2 controller, the CAN2 controller is still marked as active; it therefore needs to be marked as deactivated and then it must be reset (this is done in block C). Afterwards, the routine checks whether the just deactivated controller was the transmission controller (decision G). In the example (Figure 6.7), the CAN2 controller is not the transmission controller, the result of the check is therefore negative. Thus, no further actions are required by the CAN2 qua routine ISR and it simply finishes. Finally, the previously preempted tx request routine continues its execution and finishes (this is shown at the end of Figure 6.7).

6.4. Fault-tolerance capacities of the media management driver

A ReCANcentrate node that uses the media management driver can tolerate the failure of one of its links or the crash of one of its CAN controllers in a manner that is transparent to the user application it executes. In addition, the media management driver allows to tolerate to some extent the CAN inconsistency scenarios described in Section 3.4.3. The next two subsections describe how these inconsistency scenarios are tolerated.

6.4.1. Tolerance of the inconsistent message omission scenario identified by Rufino et al.

As described in Section 3.4.3, in the inconsistent message omission scenario that Rufino et al. [1998] identified for CAN, some nodes receive a copy of a frame, whereas others do not receive any copy at all. More specifically, the frame is initially transmitted and then rejected by some CAN nodes because they detect an error in the next-to-last bit of the EOF; whereas it is accepted by others because they detect a dominant bit in the last bit of the EOF and CAN's last-bit rule applies. After the initial transmission, the transmitter schedules a retransmission, but none of the nodes receive the retransmitted frame because the controller of the transmitting node fails before the retransmission. Thus, the nodes that did not get the frame initially do not get the frame at all.

In contrast, in ReCANcentrate each node gets at least one copy of the frame thanks to the media management driver. To explain how this works, note that the scenario that Rufino et al. identified for CAN can be divided into two separate cases because in ReCANcentrate each node has two CAN controllers.

Case 1

The first case is one where none of the controllers of the transmitting node notify of the delivery event. Figure 6.8 illustrates this case and shows how the inconsistent message omission is avoided.

A set of receivers X does not have a controller that notifies the reception of the initial transmission because both its controllers detect an error in the next-to-last bit of the EOF. Thus, both CAN controllers of these nodes reject the initial copy of the frame and do not notify its reception. The application running on a node X does therefore not receive the initial copy of the frame.

On the other hand, two other sets of receivers, Y and Z, have at least one CAN controller that does notify a reception. These controllers notify because they do not detect an error in the next-to-last bit of the EOF, but instead detect a dominant bit during the last bit of the EOF. This dominant bit is the first dominant bit of the error frame transmitted by the other controllers. Because it falls within the last bit of the EOF, the controllers are obliged to accept the frame (see Section 3.4.3) and they consequently notify the reception.

The fact that the two controllers of a receiver Y notify the reception means that for these receivers the scenario is equivalent to a fault-free reception (see Section 6.3.1). Thus, each application running on a receiver Y receives the initial copy of the frame.

Each application running on a receiver Z does also receive the initial copy of the frame. To better understand why, refer to the flowchart of the rx routine, which is depicted in Figure 6.2. In

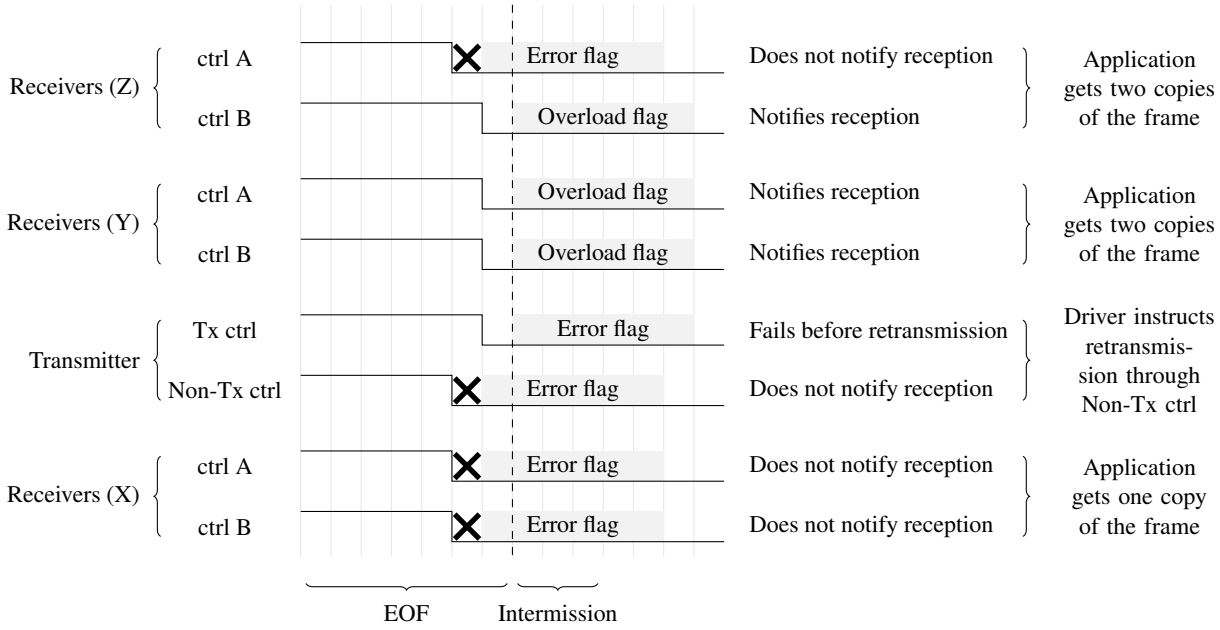

Figure 6.8.: Management of the inconsistent message omission scenario identified by Rufino et al. in ReCANcentrate (case 1). In this case neither of the controllers of the transmitting node notifies of the delivery event. The inconsistent message omission is avoided because the driver executing on the transmitter ensures that a retransmission occurs through the non-faulty controller. Thus, the set of receivers X, which did not receive a copy of the initially transmitted frame, do get a copy when the retransmission occurs.

particular, note that the rx routine ISR corresponding to the notifying controller (ctrl B of set Z in Figure 6.8) is invoked by that controller's CAN event tracker ISR and then manages the delivery event alone. When this happens, as Figure 6.2 shows, the rx routine resets the tracking variable previously set by the CAN event tracker ISR (block B); determines that the frame was not yet managed (decision E); waits K units of time (block F); determines that the other controller did not notify a transmission (decision I), that there is no transmission pending (decision L), and that the other controller did not notify a reception (decision N); copies the received frame from the controller's hardware reception buffer to the driver reception buffer (block O); signals a successful reception to the user application (block M); and releases the reception buffer of the controller (block C).

The summary so far is that there are nodes that did not get the initial copy of the frame (set X) and nodes that did get it (sets Y and Z). Moreover, the transmission controller of the transmitting node fails and it therefore does not retransmit. Remember, this is what caused an inconsistent message omission in CAN. In ReCANcentrate, however, no inconsistent message omission occurs. This is so because the media management driver instructs the retransmission of the frame through the transmitting node's non-faulty controller, which initially was the non-transmission controller. Specifically, the retransmission occurs because of the following.

Before the transmitter initiated the transmission of the frame, the transmission timer was enabled by the tx request routine. Then, when the transmitter was sending the EOF of that frame, its non-transmission controller detected an error in the next-to-last bit of the EOF and the transmission controller detected a dominant bit in the last bit of the EOF (see Figure 6.8). As a result, neither of the controllers of the transmitter notified the delivery event. Thus, none of the media management routines is executed on the transmitter for that delivery event. Moreover, since the transmission controller fails, it will not be able to retransmit. This means that the faulty transmission controller will not cause a second delivery event for the pending frame and therefore no routine will be invoked to disable the transmission timer. The consequence is that the transmission timer expires, which invokes the qua routine for the transmission controller.

When the qua routine is invoked on the transmitter, it does the following. It determines that the controller for which it was invoked, the transmission controller, is marked as active (decision B in Figure 6.3); marks the controller as inactive and resets it (block C); determines that the controller that is being quarantined is the transmission controller (decision G); disables the transmission timer (block F); determines that the other controller is still marked as active (decision E); marks the other controller as the new transmission controller (block D); determines that there is a transmission still pending (decision H); determines that the quarantined controller did not notify a transmission (decision J); and instructs the retransmission of the pending frame through the new transmission controller (block K).

Thanks to the retransmission instructed by the qua routine, another copy of the frame is transmitted. This second copy will hopefully be received by the set of receivers X, thereby ensuring that they also get a copy of the frame. In this way, all receivers will have received at least one copy and the inconsistent message omission is avoided.

Case 2

The second way in which the inconsistent message omission scenario identified by Rufino et al. can manifest itself in ReCANcentrate is illustrated in Figure 6.9.

The set of receivers X, Y, and Z act as described in case 1 (Figure 6.8). The difference is in the transmitter. Now (in Figure 6.9), the transmitter's non-transmission controller *does* notify a reception because that controller *does not* detect an error in the next-to-last bit of the EOF. As a consequence, the CAN event tracker ISR for the non-transmission controller is invoked, which in turn invokes the rx routine.

The rx routine on the transmitter does the following. It resets the tracking variable previously set by the CAN event tracker ISR (block B in Figure 6.2); determines that the frame was not managed yet (decision E); waits K units of time (block F); determines that the other controller did not notify a transmission (decision I), that a transmission is pending (decision L), and that the received frame was the one pending to be transmitted (decision K); increases the omission counter (block J); and releases the reception buffer of the non-transmission controller (block C).

Notice that the transmision timer is not reset by the rx routine and that since the transmission controller is faulty, no routine associated with the pending transmission will execute. The result is thus the same as before: the transmission timer expires, thereby triggering the qua routine. The qua routine then instructs the retransmission of the frame trough the surviving controller, which allows the set of receivers X to also receive a copy of the frame.

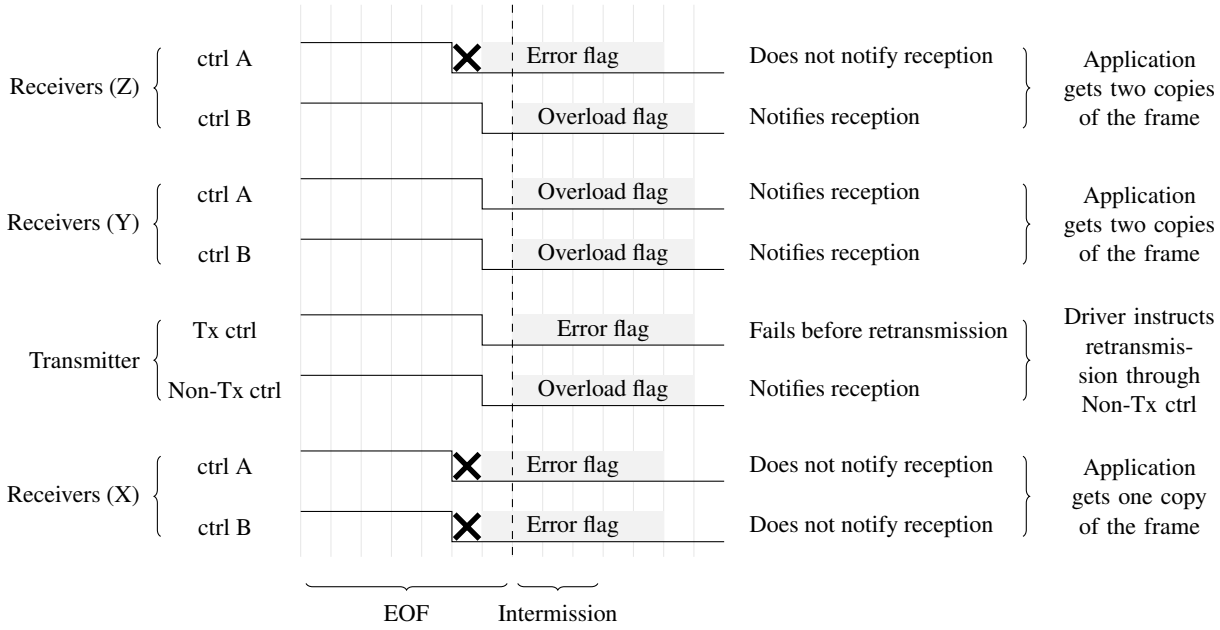

Figure 6.9.: Management of the inconsistent message omission scenario identified by Rufino et al. in ReCANcentrate (case 2). As opposed to case 1 (see Figure 6.8), in this case the non-transmission controller of the transmitting node does notify the reception of the frame. However, just as in case 1, the inconsistent message omission is avoided because the driver executing on the transmitter ensures that a retransmission occurs through the non-faulty controller. Thus, the set of receivers X, which did not receive a copy of the initially transmitted frame, do get a copy when the retransmission occurs.

6.4.2. Tolerance of the inconsistent message omission scenario identified by Proenza and Miro-Julia

As described in Section 3.4.3, the inconsistent message omission scenario that Proenza and Miro-Julia identified for CAN is identical to the one identified by Rufino et al., except that the transmitting controller omits the retransmission not because it fails but because it detects an additional error. This additional error causes the transmitting controller not to detect the first bit of the error frame transmitted by the other controllers, thereby causing it to consider the transmission as successful. Having considered the transmission as successful, it does not retransmit the frame. Thus, the controllers that detected an error in the next-to-last bit, which did not receive the frame initially, do not receive any copy of the frame.

In ReCANcentrate, the scenario that Proenza and Miro-Julia identified for CAN can also manifest itself in two different ways. We therefore also consider two cases for this scenario. However, as opposed to the scenario identified by Rufino et al., where both cases were tolerated, of the two cases for the scenario identified by Proenza and Miro-Julia only the first is tolerated by the media management driver.

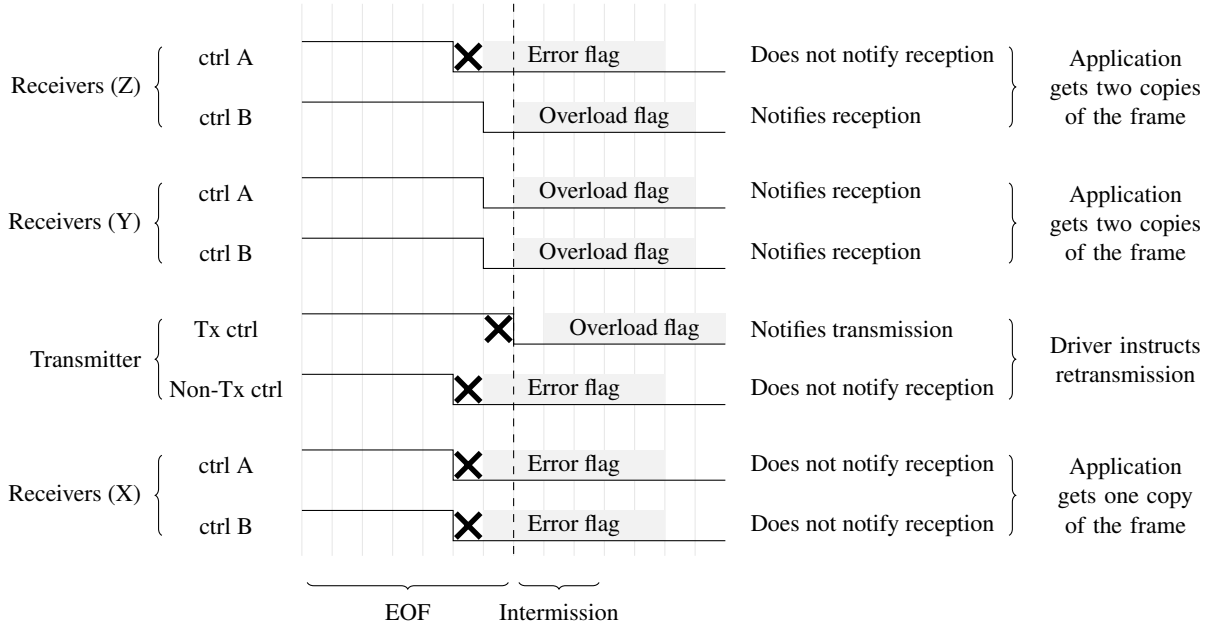

Figure 6.10.: Case where the inconsistent message omission scenario identified by Proenza and Miro-Julia is avoided in ReCANcentrate. The inconsistent message omission is avoided because the driver executing on the transmitter ensures that a retransmission occurs through the transmission controller. Thus, the set of receivers X, which did not receive a copy of the initially transmitted frame, do get a copy when the retransmission occurs.

Case 1

Figure 6.10 illustrates the case where the inconsistent message omission identified by Proenza and Miro-Julia is tolerated by the media management driver.

As in the previous section, a set of receivers X does not get the frame initially because both its controllers detect an error in the next-to-last bit of the EOF. Also, as in the previous section, two other sets of receivers, Y and Z, have at least one CAN controller that does not detect an error in the next-to-last bit of the EOF. These controllers accept the initially transmitted copy of the frame.

The set of receivers X, Y, and Z act as in the two cases described in the previous section. Moreover, like in that section, the transmitter retransmits, but due to a different reason. Specifically, this time the transmission controller of the transmitter notifies of a transmission. This invokes the CAN event tracker ISR for the transmission controller, which in turn invokes the tx routine.

The tx routine does the following. It resets the tracking variable that the CAN event tracker ISR set and disables the transmission timer (Figure 6.1, block B); it determines that the frame was not managed yet (decision D); it waits K units of time (block E); it determines that the other controller did not notify of a reception (decision H) and that the omission counter has not reached its maximum value (decision I); and it increases the omission counter and instructs

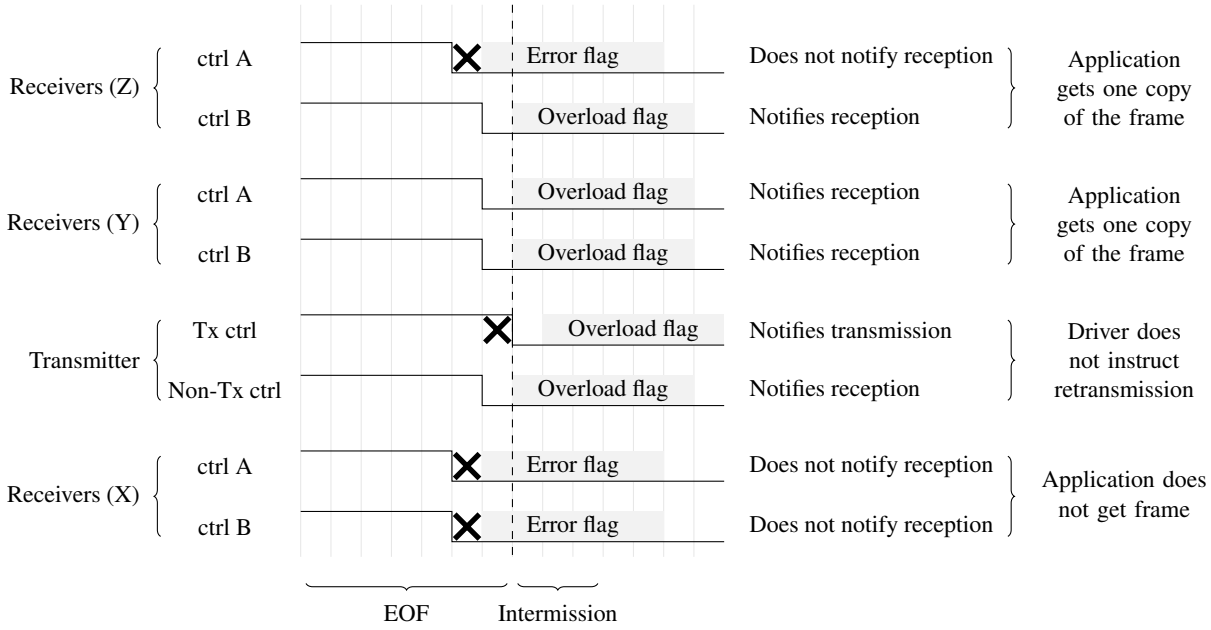

Figure 6.11.: Case where the inconsistent message omission scenario identified by Proenza and Miro-Julia is *not* avoided in ReCANcentrate. The inconsistent message omission is not avoided because the driver executing on the transmitter does not instruct a retransmission. Thus, the set of receivers X do not get a copy of the frame.

a retransmission (block J). Note that if the omission counter had reached its maximum value, the transmitter would consider the transmission as successful (block F). This is so because it is highly improbable that an inconsistency scenario had occurred a number of consecutive times equal to the maximum value of the omission counter. So, if the omission counter reaches its maximum value, the cause for the non-transmission controller not notifying is almost certainly not because of an inconsistency scenario, but more likely because it has crashed or its downlink is stuck-at-recessive. In that case we must not do any further retransmissions.

Case 2

Figure 6.11 shows the other possible manifestation in ReCANcentrate of the scenario identified by Proenza and Miro-Julia. Unfortunately, in this case the inconsistent message omission is not avoided by ReCANcentrate.

The set of receivers X, Y, and Z act as in the previous cases; the transmitter, however, now receives a notification from both its controllers. This is so because the non-transmission controller does not detect an error in the next-to-last bit of the EOF but instead accepts the transmitted frame. This causes the media management driver to *not* instruct the retransmission of the frame. Specifically, there are two possible ways for this to occur.

The first possibility is that the tx routine gets invoked before the rx routine. In that case, the tx routine resets its tracking variable and disables the transmission timer (block B in Figure 6.1);

it determines that the frame was not managed yet (decision D); waits K units of time (block E); determines that the other controller did notify a reception event (decision H); marks the frame as already managed and resets the omission counter (block G); and signals a successful transmission to the application (block F). The rx routine that gets executed next, on the other hand, just resets its tracking variable (block B in Figure 6.2), determines that the frame was already managed (block E), marks the frame as no longer managed for the next delivery event (block D), and releases the reception buffer of the non-transmission controller (block C).

The second possibility is that the rx routine gets invoked before the tx routine. In that case, the rx routine resets its tracking variable (block B in Figure 6.2); determines that the frame was not managed (decision E); waits K units of time (block F); determines that the other controller notified a transmission (decision I); marks the frame as managed and resets the omission counter (block H); signals a successful transmission to the application (block D); and releases the reception buffer of the non-transmission controller (block C). Afterwards, when the tx routine starts to execute, it simply resets its tracking variable and disables the transmission timer (block B in Figure 6.1), determines that the frame was already managed (decision D), and finishes by resetting the variable that indicates that the frame was already managed (block C).

7. New ReCANcentrate hardware prototype

This chapter introduces the new ReCANcentrate prototype, which is the successor of the previous prototype built by Barranco et al. [2006a] and which was introduced in Chapter 5. We begin the chapter with a brief introduction into the wirewrap prototyping technique, which is the technique that we used to build the new prototype. Then we describe the implementation of the hubs and the nodes of ReCANcentrate in the new prototype. Finally, we describe how we tested the hardware of the prototype.

7.1. The wirewrap prototyping technique

Wirewrapping is a fast prototyping technique with which prototypes can be built that are very easy to change later on. Wirewrapping is therefore particularly well suited for prototypes whose final design is not yet clear and which might change in the future. Moreover, wirewrap prototypes are very easy to repair.

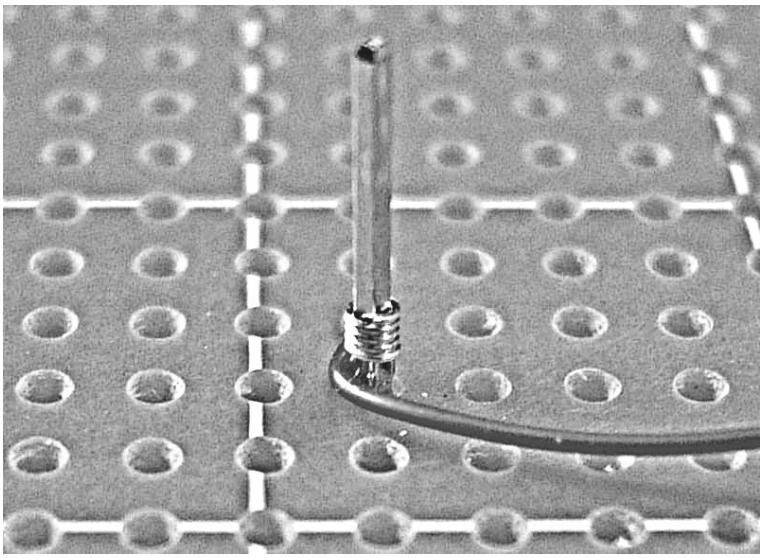

Figure 7.1.: Pin with a wire connected to it through wirewrapping. (Source http://upload.wikimedia.org/wikipedia/commons/thumb/3/35/Wire_Wrapping.jpg/800px-Wire_Wrapping.jpg, licensed under the public domain.)

To built a prototype using the wirewrapping technique one uses a ready-made perforated board

onto which electronic components can be mounted such as resistors, capacitors, and dual in-line package (DIP) integrated circuits. Although the electronic components can be often mounted directly on the perforated board, usually they are mounted on sockets that are then directly attached to the board instead. These sockets have long pins that have a square cross section especially suited to wind wires around them.

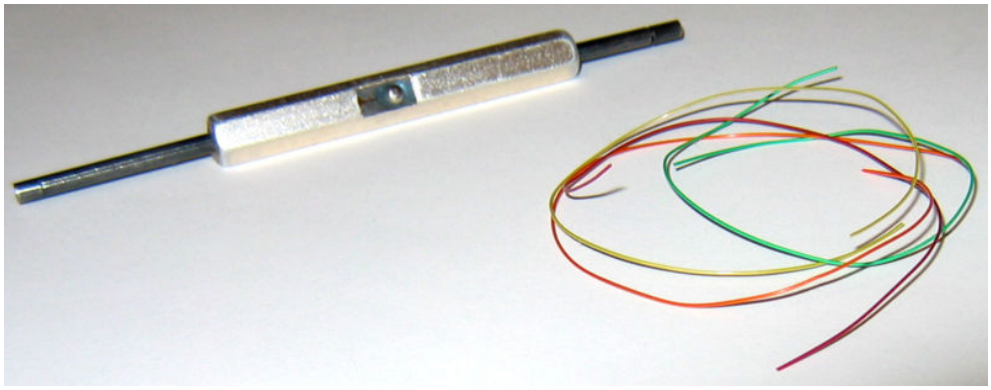

Figure 7.2.: Manual wirewrapping tool and wirewrapping wire. (Source http://upload.wikimedia.org/wikipedia/commons/thumb/f/f5/Manual_wire_wrap_tool_and_wire_wrap_wire_in_various_colours.jpg/800px-Manual_wire_wrap_tool_and_wire_wrap_wire_in_various_colours.jpg, licensed under the public domain.)

More specifically, connections are made by winding thin wire—whose ends have been stripped off their insulating plastic—around the to-be-connected pins. For this, special tools known as wirewrapping tools are used. Figure 7.1 shows the pin of a socket to which a wire has been connected through wirewrapping. Figure 7.2 shows a manual wirewrapping tool together with some wirewrapping wire. The tool is basically a thin hollow cylinder with an opening and a second hole on the side. The stripped wire is placed into the hole on the side and the opening of the tool is placed onto the to-be-connected pin. The tool is then turned so that the wire is wound around the pin. This connects one end of the wire to the pin. The other end is connected to a second pin analogously, thereby creating a connection between the first and the second pin.

7.2. Implementation of the ReCANcentrate hubs

Figure 7.3 shows a sketch of the hardware of one of our prototype's hubs. We built each ReCANcentrate hub on a perforated wirewrapping board. Onto each of these boards we mounted an XSA-3S1000 prototyping board from XESS [XESS Corporation, 2007], which contains an FPGA manufactured by Xilinx.

The particular FPGA included in the XSA-3S1000 is a Xilinx *Spartan 3 XC3S1000* [XESS Corporation, 2007]. It is the same FPGA model that Barranco et al. [2006a] had used to implement the hubs in their prototype. We were therefore able to reuse for our FPGAs the VHDL code that Barranco et al. had written for their prototype—as mentioned in Chapter 5, this VHDL

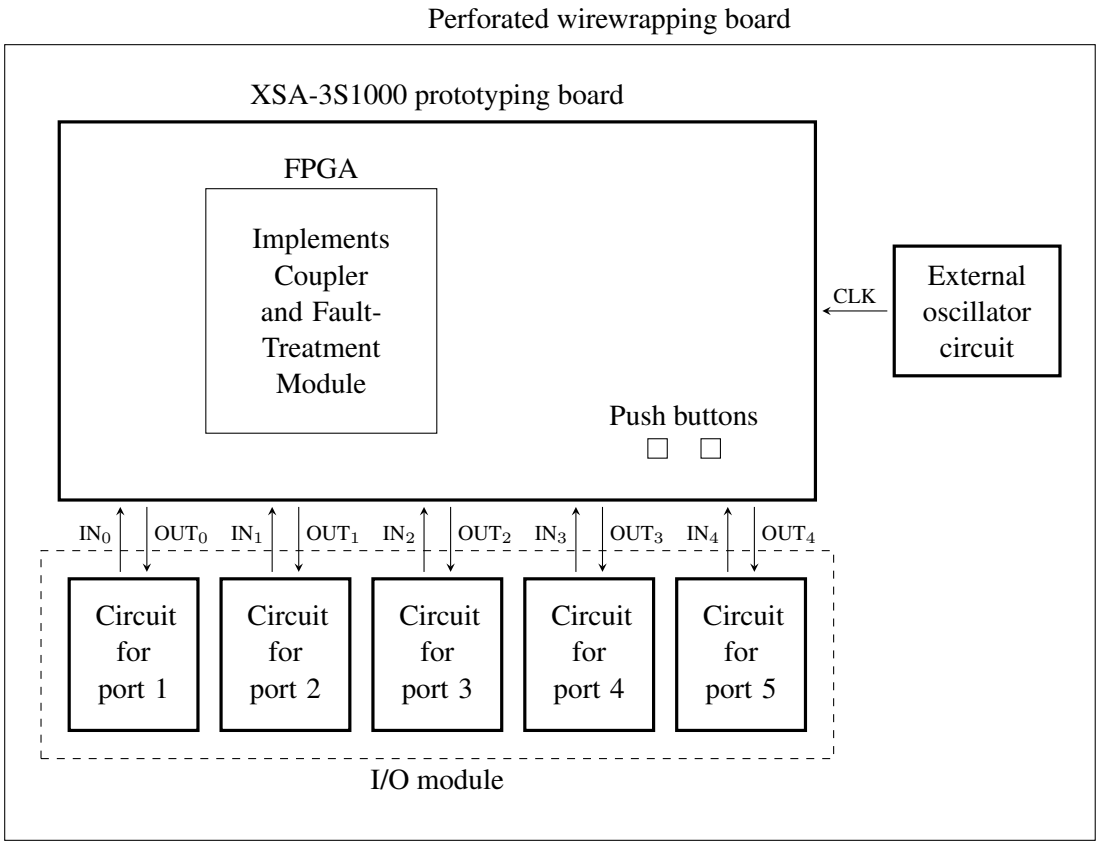

Figure 7.3.: Implementation of a ReCANcentrate hub, showing its main building blocks. We implemented each hub using a perforated wirewrapping board onto which we mounted an XSA-3S1000 prototyping board, whose FPGA implements the hub's coupler and fault-treatment modules; 5 electronic circuits implementing ports of the hub's I/O module, 3 of which are used by the FPGA as link ports to nodes and 2 of which are used as interlink ports to the other hub; and one electronic circuit to attach an external oscillator to the FPGA.

code implements the coupler module and the fault-treatment module of a ReCANcentrate hub.

Apart from a coupler module and a fault-treatment module, we also had to implement an I/O module for each hub. As Figure 7.3 shows, the I/O module is comprised of 5 electronic circuits implementing a port of the I/O module each. Each of these 5 electronic circuits implements the circuit shown in the schematic of Figure 5.8 from Chapter 5 (page 56).

The schematic is explained in more detail in Section 5.4.4, but to summarize, each I/O module port is comprised of an RJ45 plug and two PCA82C250 CAN transceivers (one for the incoming sublink or uplink, and one for the outgoing sublink or downlink); UTP Cat 5 Ethernet cables are used for the links and interlinks; and the wires labeled IN and OUT in the schematic (also shown in Figure 7.3) are the points to be connected to the coupler and fault-treatment module.

Focusing on the last point, that the IN and OUT wires have to be connected to the coupler and fault-treatment module, note that this means that the IN and OUT wires of each port have to be

connected to I/O pins of the FPGA contained within the XSA-3S1000 prototyping board. For this purpose, we used the 84-pin prototyping header located at the bottom of the XSA3S1000 [XESS Corporation, 2007]. Specifically, the IN and OUT wires of the hub ports were connected to a subset of the pins of the header, namely, to some of the pins that are directly connected to the I/O pins of the FPGA.

Now we may ask, how does the FPGA know to which of its I/O pins we connected the IN and OUT wires? The answer is that the FPGA pins to which the IN and OUT wires are connected are mapped to appropriate variables in Barranco et al.'s VHDL code. This mapping is specified in a so called *user constraints file* (UCF), which is taken into account by the tools used to program an FPGA. The relevant section of the UCF file has the following contents (the complete file is found in Appendix E):

```
NET "rx_0" LOC = "H15" ;
NET "rx_1" LOC = "H14" ;
NET "rx_2" LOC = "G12" ;
NET "rehRx_0" LOC = "G16" ;
NET "rehRx_1" LOC = "H13" ;

NET "tx_0" LOC = "G15" ;
NET "tx_1" LOC = "G14" ;
NET "tx_2" LOC = "F14" ;
NET "hubTx_0" LOC = "F15" ;
NET "hubTx_1" LOC = "G13" ;
```

The first line maps a pin labeled H15 to a variable named `rx_0`, the second line maps a pin labeled H14 to a variable named `rx_1`, the third line maps a pin labeled G12 to a variable named `rx_2`, and so forth for the remaining lines. Variables `rx_0`, `rx_1`, and `rx_2` represent uplinks; variables `tx_0`, `tx_1`, and `tx_2` represent downlinks; variables `rehRx_0` and `rehRx_1` represent incoming sublinks; and variables `hubTx_0` and `hubTx_1` represent outgoing sublinks.

As an example, and considering the above listed section of the UCF file, the wire labeled IN_0 in Figure 7.3 is connected to pin H15 and is therefore mapped to variable `rx_0`. This means that IN_0 is interpreted by the coupler and fault-treatment modules implemented in the FPGA as the contribution received from the node whose uplink is connected to the first port of the hub. Similarly, the wire labeled OUT_0 in Figure 7.3 is connected to pin G15 and is therefore mapped to variable `tx_0`. Thus, this pin is used to send the signal coupled by the hubs (B_T in Figure 5.3, page 45) to the node whose downlink is connected to the first hub port. The remaining IN and OUT wires of Figure 7.3 are mapped analogously to the remaining variables in the UCF file.

Regarding the external oscillator circuit, shown on the right of Figure 7.3, it implements the electronic circuit of the schematic of Figure 5.9 (Chapter 5, page 57). As mentioned in Section 5.4.4, the oscillators that the hubs and the nodes use must have the same frequency, or at least be a multiple of each other (which can then be scaled up or down), so that all hubs and nodes can sample each other's transmitted bits. Unfortunately the oscillators provided with the hubs' XSA-3S1000 prototyping boards have a frequency of 100 MHz [XESS Corporation, 2007], whereas the oscillators available to a node have frequencies of 7.37 MHz, 512 KHz, and 32.768 KHz [Microchip, 2006c]—none of which is a multiple of 100 MHz. To overcome the mismatch between

these frequencies, we added to each hub an external oscillator of 14.74 MHz—which is a multiple of a node's 7.37 MHz oscillator—using the external oscillator circuit. This circuit provides a wire carrying a clock signal (labeled CLK in Figure 5.9 and in Figure 7.3) which is connected to a dedicated clock input pin of the XSA-3S1000 board's prototyping header.

7.3. Implementation of the ReCANcentrate nodes

The first step in implementing the nodes for the new prototype was to decide what hardware components to use. The hubs of the new prototype were basically the same as the hubs of the previous prototype. We therefore chose the same hardware for them. In contrast, the hardware requirements for the nodes of the new prototype (see Section 5.3.4) were different and they therefore required different hardware. The *dsPICDEM prototyping board* from Microchip [Microchip, 2006c], which contains a *dsPIC30F6014A microcontroller* [Microchip, 2006a], satisfies all of these requirements:

- The dsPIC30F6014A microcontroller has two embedded CAN controllers.
- The interrupts in a dsPIC30F6014A microcontroller can be nested, that is, any interrupt service routine that is in progress may be interrupted by another source of interrupt with a higher user assigned priority level [Microchip, 2006b, Section 6].
- The dsPIC30F6014A microcontroller has seven user selectable priority levels for the interrupts. Moreover, if two interrupts have the same user selectable priority and are both pending, then, to resolve the priority conflict, a so-called *natural order priority* is used. According to this priority, interrupts with a lower address in the interrupt vector table have a higher priority than interrupts with the same user selectable priority that have a higher address in the interrupt vector table.
- Interrupts can be generated through software in the dsPIC30F6014A microcontroller. The microcontroller has several interrupt flag status registers whose bits indicate if a given interrupt is pending. Simply setting such a bit to the appropriate logic value will generate the interrupt corresponding to that bit. However, all interrupts correspond to hardware peripherals and there are no special purpose software interrupts; luckily, interrupts corresponding to unused peripherals can be used as software interrupts.
- The CAN controllers embedded in the dsPIC30F6014A microcontroller provide a so-called error warning interrupt. If this interrupt has been enabled, then, whenever a controller's receive error counter (REC) or transmission error counter (TEC) reaches the value of 96, known as the error warning limit, an interrupt is generated. Note that the value of the error warning limit, 96, is below 128, the value at which a controller enters the error-passive state.
- The CPU in the dsPIC30F6014A microcontroller is fast enough to execute the driver's routines before the next delivery event takes place.

That last point requires a little more elaboration. The microcontroller has three internal oscillators of frequencies 32.768 kHz, 512 kHz, and 7.37 MHz. Moreover, it is possible to connect external oscillators. The basis for the instruction clock is the frequency of one of these oscillators, which can then be scaled up using a phase locked loop (PLL) or scaled down using a post-scaler. Figure 7.4 shows the microcontroller's three internal oscillators (labeled as FRC, Y2, and LPRC) and the two external oscillator slots provided with the dsPICDEM prototyping board (labeled Y1, which comes with a 7.37 MHz oscillator, and Y3, which comes with no oscillator preinstalled); moreover, it also shows the PLL and the post-scaler. The maximum oscillator frequency (F_{OSC} in Figure 7.4) that can be derived from the oscillators is 117.92 MHz, achieved when a 7.37 MHz frequency (either Y1 or FRC) is multiplied by 16 in the PLL and then divided by one in the post-scaler. However, the instruction clock frequency (F_{CY}) cannot be the maximum oscillator frequency; instead, the instruction clock frequency always results from dividing the oscillator frequency by 4. Thus, the maximum instruction clock frequency is $\frac{117.92}{4} = 29.48\,\text{MHz}$, or, stated differently, each instruction cycle is $\frac{1}{29.48\,\text{MHz}} = 33.92\,\text{ns}$ long. When we evaluated whether the CPU of the dsPIC30F6014A microcontroller satisfies our speed requirements, we came to the conclusion that this is fast enough: with a bit rate of 1 Mbps (the maximum CAN bit rate) each bit time is one microsecond long and therefore $\lfloor \frac{1\,\mu\text{S}}{33.92\,ns} \rfloor = \lfloor 29.48 \rfloor = 29$ instructions can be executed in each bit time. So, in the worst case, when the shortest possible frame is exchanged (a 44 bit remote frame or a 44 bit data frame) $44 \cdot 29 = 1276$ instruction cycles can be executed. We considered this to be enough for two main reasons. First, in the dsPIC30F6014A microcontroller "all instructions execute in a single cycle, with the exception of instructions that change the program flow, the double-word move (MOV.D) instruction and the table instructions[1]" [Microchip, 2006b, Section 5]. This means that most of the instructions of the driver would be single cycle. Second, we estimated the number of instructions that the driver would have to perform before each frame exchange to be fairly low. The reason for this is that the driver does not have to execute any CPU or memory intensive tasks (such as image processing), but only the short routines described in the previous chapter.

As we had assessed that it was viable to implement the nodes using dsPICDEM boards, we decided to use these boards for our prototype. Figure 7.5 shows a sketch of the hardware of one of our prototype's nodes.

On the left-hand side we see a node's core, implemented with a dsPICDEM board, which includes a dsPIC30F6014A microcontroller and two CAN controllers embedded in the microcontroller. Moreover, we see that a node core's dsPICDEM board contains four LEDs, two push buttons, and an 80-pin header that gives easy access to the pins of the microcontroller and the pins of the embedded CAN controllers.

On the right-hand side we see the node's I/O module, implemented on a perforated wirewrapping board. This board provides the means to connect the node to each of the hubs: a link connected to port 1 connects the node to one hub, and a link connected to port 2 connects the node to the other hub. The two electronic circuits on the I/O module implement ports equivalent to the ones on a ReCANcentrate hub, that is, they also implement the schematic of Figure 5.8 from Chapter 5. The IN and OUT wires of a given circuit port are respectively connected to the

[1] The dsPIC30F6014A CPU has a Harvard architecture, that is, it has physically separate program and data memory. Table instructions are used to transfer data between program memory space and data memory space.

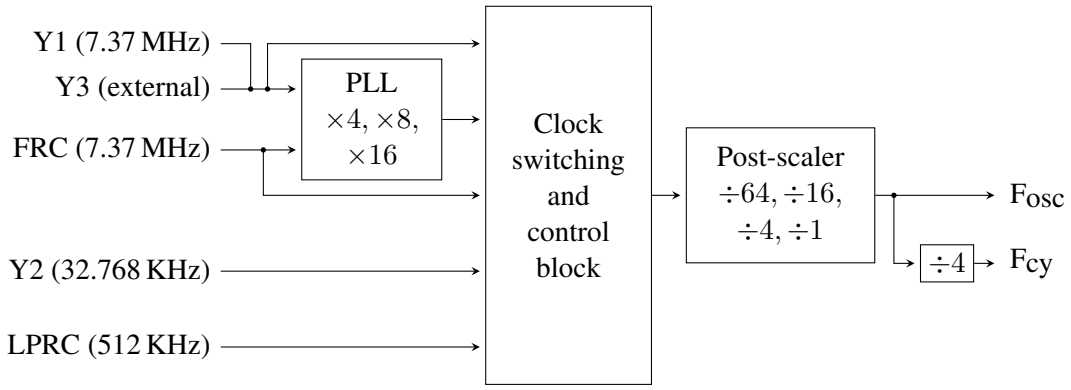

Figure 7.4.: Clock sources in a dsPICDEM prototyping board.

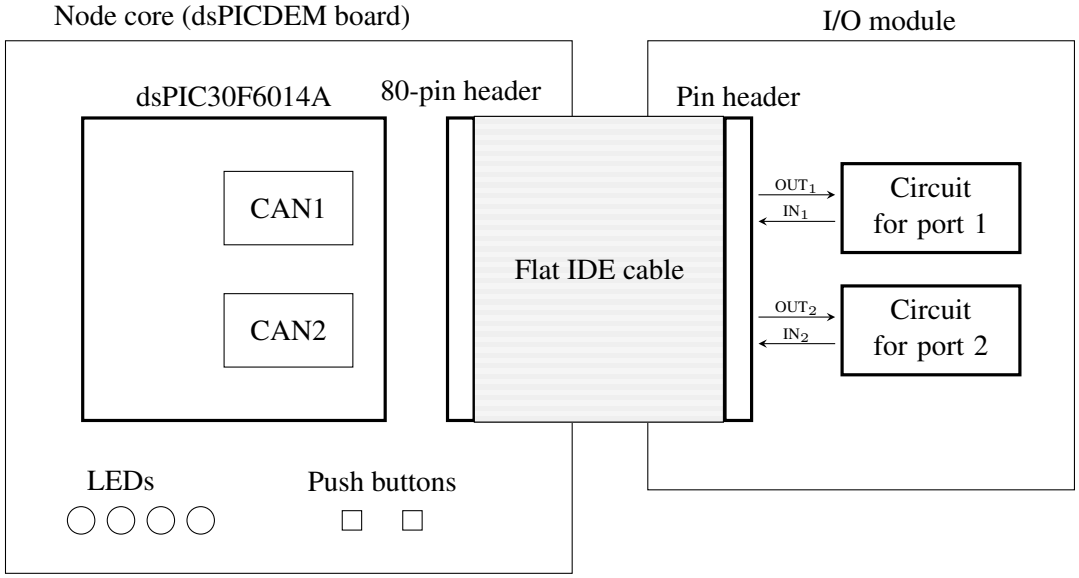

Figure 7.5.: Implementation of a ReCANcentrate node, showing its main building blocks. We implemented each node using two boards: a dsPICDEM prototyping board and an I/O module implemented on a perforated wirewrapping board.

reception and transmission pins of one of the two CAN controllers. Specifically, this is done by means of a flat IDE cable connecting a pin header on the I/O module to the 80-pin header on the dsPICDEM board. By having the CAN controllers connected to the ports of the node's I/O module in this way, the driver, which is running on the microcontroller, can transmit frames to the hubs and receive frames from the hubs.

7.4. Testing the hardware of the prototype

Once we had built the prototype's hardware, we tested it to make sure that it had been built correctly. We did this in an incremental fashion, that is, instead of testing the whole hardware at once we first tested a subset and then incrementally added more parts. This incremental approach had the advantage of making it easier to localize the cause of any errors found during the tests. If a particular test was successful, but the subsequent test was not, we could determine with high probability that the root of the problem was the part of the system that had been added last. Next we will describe each of these tests.

7.4.1. Verification of the electronic circuits

Before any other test, we first had to be sure that each pin of each hardware device—FPGA, microcontroller, oscillator, and so forth—was connected to where it ought to be and that there were no short-circuits. Moreover, we had to be sure that the electronic circuits for the I/O ports and external oscillators, which we build from the schematics shown in Section 5.4.4, exactly matched those schematics. We verified both things by using a digital multimeter in continuity test mode—the one where the multimeter beeps when there is a direct connection between the probes. Afterwards we proceeded to test the nodes, which, once we were sure that they worked, would be used to test the hardware of the hubs.

7.4.2. Testing the node cores

We started the testing of each node by first considering only one of its two building blocks, namely, the node core. As we said in Section 7.3, each node core was implemented as a dsPICDEM board. The tests therefore consisted in verifying that each dsPICDEM board was functioning correctly, that is, that they had no manufacturing defects or shipping damage.

The tests consisted of two simple programs which use the CAN controllers of a node's microcontroller in *loopback mode* [Microchip, 2006b, Section 23]. This mode connects the internal transmit signal of a CAN controller to its internal receive signal. Moreover, it causes special hardware to generate an ACK bit in order to make transmissions successful. The loopback mode therefore allowed us to test the dsPICDEM boards, the embedded dsPIC30F6014A microcontrollers, and their CAN controllers in isolation, without having to worry about any potential complications that might arise at the hubs or the I/O modules of the nodes. In fact, for these two tests the physical setup of the prototype was irrelevant: it did not matter how the hubs and nodes were interconnected through their I/O modules. Even more, the node cores did not even have to be connected to their corresponding I/O modules.

Concerning the details of the two tests, both basically initialized a node's CAN controller in loopback mode and subsequently sent a single data frame with it. This frame was thereafter received by that same CAN controller. After each stage—initialization, transmission, and reception—another one of the four LEDs on the dsPICDEM board was turned on. The only difference between the two tests was that the first used polling to determine when the transmission and reception of the frame occurred while the second used interrupts. The source code for the two tests can be found in Appendix B, sections B.2.1 and B.2.2. Finally, these two tests allowed us to learn how to properly initialize and configure the CAN controllers and how to send and receive frames using them.

After these tests, we were confident that the node cores were functioning correctly. We therefore proceeded to use them together with the second building blocks of the nodes: the nodes' I/O modules.

7.4.3. Testing the node cores together with their I/O modules

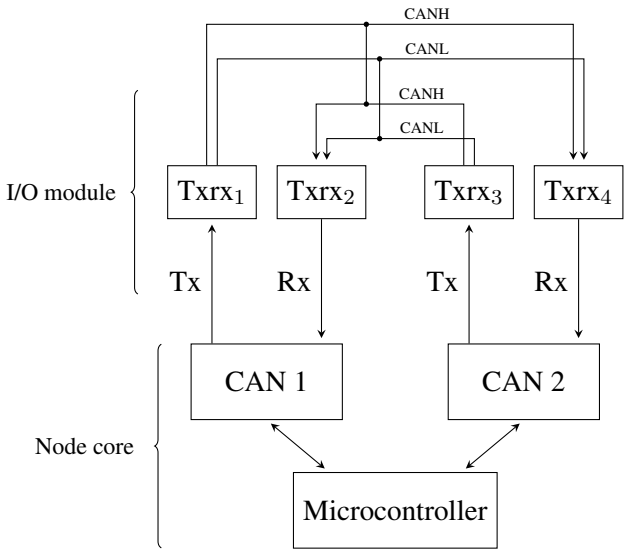

Figure 7.6.: Connecting the two CAN controllers of a single ReCANcentrate node to each other.

The next set of tests used both the node cores and their attached I/O modules. Each test consisted in taking a single node core with its respective I/O module in isolation. Specifically, each test consisted in transmitting frames between the two CAN controllers of a given node core by using the corresponding I/O module. This was accomplished by interconnecting both CAN controllers of each node as shown in Figure 7.6. We connected the transmission transceiver of the first CAN controller (Txrx_1 in Figure 7.6) to the reception transceiver of the second CAN controller (Txrx_4), and we connected the transmission transceiver of the second CAN controller (Txrx_3) to the reception transceiver of the first CAN controller (Txrx_2). Note that both transceiver interconnections are each comprised of a CANH and a CANL wire.

Remember that a transmitting CAN controller observes the medium and verifies that it receives the frame that it is transmitting (with the ACK slot overridden). If a CAN controller does not receive what it transmits, it detects an error. Therefore, to allow the communication between the two CAN controllers of a single node, we also had to ensure that each CAN controller receives its own frame. This was accomplished through a couple of additional interconnections that make the setup equivalent to a CAN bus that has both controllers attached to it. Specifically, this was achieved by connecting the CANH and CANL wire of one transceiver interconnection to the CANH and CANL wire of the other transceiver interconnection. This is also depicted in Figure 7.6. The source code we executed for this test can be found in Appendix B, Section B.3.

At this point we had all the relevant hardware of the nodes tested. We could therefore proceed to use the nodes to test the hardware of the hubs.

7.4.4. Testing the hardware of the hubs

After we verified that the hardware of the nodes worked correctly, we used them to test the hardware of the hubs, that is, the hub I/O modules and the XSA-3S1000 boards with their embedded FPGAs. We wrote a simple AND-coupling module in VHDL and downloaded it to each of the FPGAs. The VHDL source code can be found in Appendix B, Section B.4. This simple AND-coupling module takes the uplink signals of each of the five ports of a hub as inputs to an AND gate and outputs the result to the downlinks. In other words, it simply performs CAN's wired-AND functionality.

Figure 7.7 shows the setup; note that to simplify the figure, it only shows two nodes and two hub ports (transceiver pairs) of the hub I/O module, instead of the three nodes and five hub ports involved in the actual experiment. As can be seen, each node is connected by means of one of their ports to this AND-coupling module.

In the experiment, we installed on one of the nodes a simple program that transmits a frame and on the other two nodes a simple program that receives the frame; the source code for these two programs is found in Appendix B, sections B.4.4 and B.4.5.

We verified that the two receiving nodes actually received the frame sent by the transmitting node. Then we executed the test again, but now with two of the nodes plugged into the ports that had previously remained unplugged. We did this for both hubs. In this way we checked that all five ports of both hubs and the two XSA-3S1000 boards worked correctly.

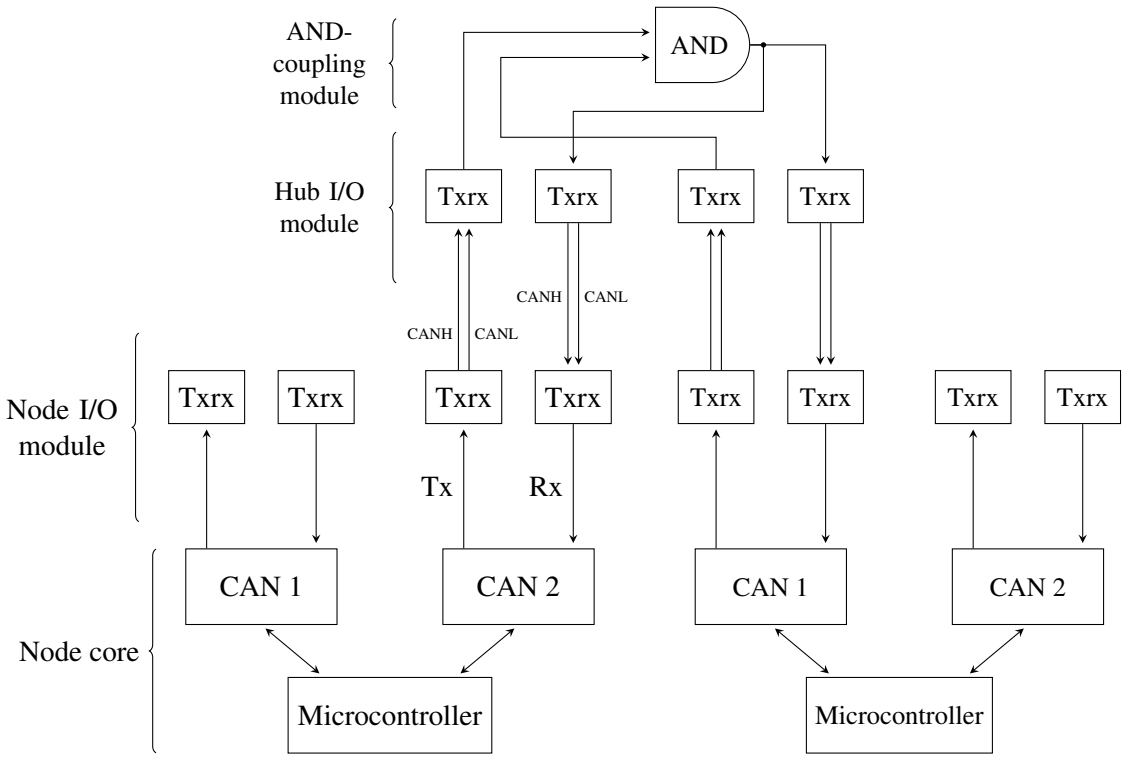

Figure 7.7.: Connecting two ReCANcentrate nodes through a signal coupler.

8. Implementation of the driver

In this chapter we describe the implementation of the media management driver. We first describe briefly the development environment that we used, then we say a few things about the methodology that we used during the development of the driver, and finally we describe the driver's source code. Note that this chapter does not describe the driver source code in excruciating detail. Instead, it just attempts to give an overview and serves as a starting point for anyone interested in the full implementation details. For the full details we refer the interested reader directly to the driver source code, which is listed in Appendix C.

8.1. Development environment

We implemented the driver using the C programming language and the toolchain provided by Microchip to program the dsPIC30F family of devices, of which the dsPIC30F6014A microcontroller which we used in our prototype is a member. This toolchain is based on the GNU compiler collection (GCC) and other GNU language tools from the Free Software Foundation (FSF). Specifically, the Microchip toolchain includes the following tools [Microchip, 2005a,b]:

- The MPLAB C30 C compiler.
- The MPLAB ASM30 assembler.
- The MPLAB LINK30 linker.
- The MPLAB LIB30 librarian/archiver.

The MPLAB C30 compiler takes as input C source code files and produces as output assembly language files. The MPLAB ASM30 assembler then takes these assembly language files as an input and produces object code from them. The object code files are then linked together using the MPLAB LINK30 linker to produce the final executable. This executable is loaded from the developer's computer to the dsPICDEM board using an ICD2 programmer, which is a small USB device manufactured by Microchip. Regarding the MPLAB LIB30 librarian/archiver, it is used to create libraries from object files. A library contains subroutines which may be needed by several programs. By linking a given library with object files from different programs, these programs can use the subroutines within the library. This allows to share code between different programs. We had no need for the MPLAB LIB30 librarian/archiver, but we used the other three tools.

Finally, in addition to the above-mentioned tools, we also used a simulator for the dsPIC30F6014A microcontroller. The particular simulator we used was the MPLAB SIM simulator from Microchip [Microchip, 2009]. This simulator models the CPU of the dsPIC30F

family of microcontrollers and several peripherals (such as timers, A/D converters, and I/O ports) of these microcontrollers. Unfortunately, however, no CAN controllers are modeled by this simulator. This was a major drawback because our media management driver uses the two CAN controllers of the dsPIC30F6014A microcontroller extensively.

Nevertheless, during the development of the driver, before it was ready enough to be executed on the physical hardware, we periodically tested the driver on the MPLAB SIM simulator. This was useful during the early stages of the development of the driver, allowing us to test parts of the driver code that did not directly depend on the existence of CAN controllers. Later on, we were even able to simulate the transmission and reception of frames in a limited manner through the use of a so-called *stimulus file* for the simulator, which is a file that tells the simulator to load specified registers, such as the registers of the CAN controllers, with specified values at specified instants of time. These stimulus files can be found in Appendix J.

8.2. Methodology

Our focus during the development of the driver was on producing readable, modular, easily modifiable, and correct code. To achieve this we used *abstract data types* (ADTs), *assertions*, and what Hunt and Thomas [1999] call the *DRY principle*.

ADTs are collections of data with a collection of operations that work on that data. ADTs have many benefits [McConnell, 2004], the following are some noteworthy ones:

- They hide implementation details. This allows a developer to change the underlying implementation without affecting the whole program.
- They make it easier to improve performance. If the implementation of an ADT is not fast enough, only the few routines of the ADT that are too slow must be rewritten instead of having to revise an entire program.
- They make programs more self-documenting and readable. By using appropriate and informative names for the operations of an ADT, the intent of source code statements becomes clearer. For instance, `light_LEDs()` is much clearer than `PORTD = 0xFFFF`, assuming that the port labeled D is the one where the LEDs are connected to and that the hexadecimal value 0xFFFF is the correct value to transmit through the port to light the LEDs.
- They make programs more obviously correct. For instance, if the program should light the LEDs, the developer is much less likely to introduce a mistake if all that has to be done is call the function `light_LEDs()` than if the developer has to determine which port to access and what value to transmit through that port every time the LEDs should be lighted.
- They make it easier to introduce changes. Instead of having to change a great number of program statements spread out around the whole program, changes are localized to the part of the program where the ADT's operations are defined.

- They allow the developer "to work in the problem domain rather than at the low-level implementation domain" [McConnell, 2004]. For instance, the developer can work with ADTs representing LEDs instead of having to work with ports and bits transmitted through those ports.

Another main technique we used during the development of the driver are assertions. Assertions are a technique often used in an approach to programming commonly known as *defensive programming* [Hunt and Thomas, 1999; McConnell, 2004]. Specifically, assertions are pieces of code that actively check whether specific assumptions hold at a given point of a program. Assertions can therefore be used to check whether assumed preconditions and postconditions actually hold. If an assertion is true, that means that the checked assumption holds in the code; if an assertion is false, that means that the code is not behaving as expected. Assertions are usually implemented as routines or macros that take at least one argument, which is a boolean expression describing the assumption to be checked. Additional arguments to an assertion may be, for instance, the message to display when the assertion fails. As an example of an assertion, consider a developer who assumes that at a given point in the program a specific variable is positive. To actively check whether that assumption is true the developer can use an assertion:

```
ASSERT(variable > 0);
```

If the variable is positive, the program continues unaffected (except for the small overhead of the assertion). However, if the variable is negative or zero, the assertion fails and the program's execution is halted. Halting the program execution is an application of another defensive programming technique which states that it is better to crash early than to crash late [Hunt and Thomas, 1999]: it is easier to identify the source of an error if the program halts as soon as the fault occurs than if the program crashes later on, when it has propagated to other parts of the program and finally manifests.

Assertions are usually only left in the code during development. Once development has finished, the developer may choose to eliminate them before the compilation to generate the final executable. In this way, the final executable will not have the overhead of the assertions.

Finally we want to highlight the DRY principle, which we also applied during the development of the driver. The DRY principle states that

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system [Hunt and Thomas, 1999].

The principle is called DRY because DRY stands for *Don't Repeat Yourself*. The DRY principle does not only apply to code but also to other contexts such as design, documentation, administrative tasks, testing, and others. This chapter, however, is only concerned with the application of the DRY principle to code, that is, to the avoidance of code duplication. Avoiding code duplication has many benefits. As McConnell [2004] put it:

> With code in one place, you save the space that would have been used by duplicated code. Modifications will be easier because you'll need to modify the code in only one location. The code will be more reliable because you'll have to check

only one place to ensure that the code is right. Modifications will be more reliable because you'll avoid making successive and slightly different modifications under the mistaken assumption that you've made identical ones.

8.3. Overview of the driver source code

In this section we overview the driver source code, which can be found in Appendix C.

8.3.1. Implementation of assertions

We implemented assertions as a C preprocessor macro. This macro can be found in the file assert.h, which is listed in Appendix C, Section C.2. To make the following discussion easier to follow we reprint the macro here:

```
#ifdef NDEBUG

#define ASSERT(expr) ((void)0)

#else

void aFailed(
        char *file_name,
        int line
);

#define ASSERT(expr) if (expr) {/*Do nothing*/} else\
        aFailed(__FILE__, __LINE__)

#endif /* NDEBUG */
```

The first thing that the macro does is to check whether the constant `NDEBUG` has been defined (NDEBUG is shorthand for "no debugging"). If it has been defined, all assertions spread throughout the code will be compiled into an empty statement, that is, a statement that does nothing (the statement consists of a simple `0`, which does nothing, cast to a void pointer in order to make the compiler complain when the developer attempts to assign that `0` to a variable); whereas if it has not been defined, all assertions will be compiled into an if-else statement. This if-else statement uses in the if-part the expression passed as a parameter to the assertion. If the expression is true, the assertion macro does nothing. On the other hand, if the expression is false, the helper function `aFailed()` is called. This function is defined in the file assert.c, listed in Section C.1 of Appendix C. The function takes two parameters: the name of the source code file where the assertion has failed and the line number within that file where the assertion has failed. The values of these two parameters are provided by the C preprocessor, that is, before the file is compiled, the preprocessor processes the file and substitutes all occurrences of `__FILE__` with a string that contains the filename of the processed file, and it substitutes all occurrences of

__LINE__ with the current line number within the file. The function `aFailed()` then copies these two values to the data memory of the microcontroller and then enters an infinite loop in which the four LEDs of the dsPICDEM board are continuously flashed. This allows for easy debugging and validation of the code: if the four LEDs are flashing, we know that the code is not behaving as expected and by simply inspecting the microcontroller's data memory (Microchip's ICD2 programmer allows in-circuit debugging) we can assess where in the code—in which file and at which line—an assumption was violated.

We used these assertions heavily in the driver code. This introduced a notable penalty on the performance of the driver; however, the penalty was not so big as to make the driver too slow to process a delivery event before the next frame is exchanged (see Section 9.2). Anyway, the assertions can easily be removed during compilation. It is possible to define a preprocessor constant by passing a command line option to the MPLAB C30 compiler [Microchip, 2005a]. Thus, to compile the driver with or without assertions is simply a matter of defining or not the constant `NDEBUG` when invoking the compiler from the command line and does not require any changes to the source code itself.

8.3.2. Abstract Data Types

The CAN frame ADT

We implemented a simple ADT for CAN frames that basically consists of a data structure with three elements: the data transmitted in a frame, the length in bytes of that data, and the identifier of the frame. The operations provided by the CAN frame ADT are a function to compare two frames and assess whether they are equal or not, a function to copy all the contents (data, data length, and identifier) of one frame to another frame, and a function that serves both to copy the data field of a frame to a buffer and to copy the contents of a buffer into a data field of a frame. The ADT is implemented in the files can_frame.c, listed in Section C.5 of Appendix C, and can_frame.h, listed in Section C.6 of Appendix C,

The CAN controller ADT

We implemented an ADT for the CAN controllers of the dsPIC30F6014A microcontroller. The source code for this ADT is found in the files can_controller.c, listed in Section C.3 of Appendix C, and can_controller.h, listed in Section C.4 of the same appendix. But before we explain this ADT and its implementation we should perhaps first explain the basic characteristics of the CAN controllers.

Each of the two CAN controllers of a dsPIC30F6014A microcontroller has a total of 70 registers associated with it. These registers can be grouped as follows:

CAN module control and status registers. There is one of these registers for each of the two CAN controllers. They are used to configure, send commands, and check the state of the corresponding CAN controller. For instance, they are used to choose whether the CAN clock (that is, the clock from which the time quantum length is derived) should be the oscillator clock (F_{OSC} in Figure 7.4, page 87) or the instruction cycle clock (F_{CY} in Figure 7.4), to choose the CAN controller operation mode (normal operation, disable mode, loopback mode, listen only mode, configuration mode, or listen all messages mode), to

indicate whether the CAN controller should be stopped when the dsPIC30F6014A microcontroller enters the idle mode, and to instruct the CAN controller to abort all pending transmissions.

Transmission buffer registers. Each of the two CAN controllers has three transmission buffers. Moreover, each transmission buffer is actually a collection of registers: for each transmission buffer there are four data field registers, each of two bytes, that hold the payload of a frame that is to be transmitted; there is a register to hold a standard identifier (in case the frame that is to be transmitted is a standard frame) and a register to hold an extended identifier (in case the frame that is to be transmitted is an extended frame); there is a register that holds the data length code, which is also used to indicate whether the frame that is to be transmitted is a remote frame; and there is a transmit buffer status and control register. The transmit buffer status and control register is used to request the transmission of the frame that is to be transmitted and to assign a priority to that frame. The priority is relative to the other transmission buffers and should not be confused with the priority on the CAN medium, which inherently comes from the identifier used for the frame. Moreover, the transmit buffer status and control register also indicates whether a message has been aborted, whether it has lost the CAN arbitration when a transmission was attempted, or whether an error occurred during a transmission.

Reception buffer registers. In addition to the transmission buffers, each CAN controller also has its own reception buffers. Specifically, there are two reception buffers per CAN controller. Just like the transmission buffers, the reception buffers are also a collection of registers: for each reception buffer there are four data field registers, each of two bytes, that hold the payload of a frame that has been received; there is a register that holds the standard identifier of a received frame and a register that holds the extended identifier of a received frame (whether the frame was extended or standard is indicated in a flag bit of the standard identifier register); and there is a register that holds the data length code of the received frame, which additionally indicates whether the frame was a remote frame or a data frame. In addition to these registers, there are a series of registers associated with each reception buffer that control whether a received frame should be accepted or discarded. These registers are the acceptance filters and the acceptance filter masks. The message acceptance filter masks indicate which bits of a received frame's identifier should be examined to decide whether to accept the received frame or not; whereas the message acceptance filters are used to determine what value those bits must have for the frame to be accepted. Finally, each of the two reception buffers has its own status and control register. These status and control registers have a flag that indicates whether the corresponding reception buffer is full, a flag that indicates whether a remote frame was received, and a flag that indicates which acceptance filter enabled the reception of a frame at the corresponding reception buffer.

Bit rate configuration registers. There are two bit rate configuration registers for each CAN controller. Together they allow to configure the bit rate at which the corresponding CAN controller should work. The first bit rate configuration register allows to set a value for the resynchronization jump width (SJW) and to set the value by which the baud rate prescaler

should pre-scale the CAN clock in order to have the correct time quantum length for a desired bit rate. The second bit rate configuration register allows to program the length of the different CAN bit time segments, as well as to choose whether the channel should be sampled once or three times. Note that having three sample points is not specified in the CAN specification [Bosch GmbH, 1991], but it is nonetheless an available option in the CAN controllers of the dsPIC30F family of devices [Microchip, 2006b].

Transmission/reception error counter registers. For each CAN controller there is a single register that contains both the CAN transmission error counter (TEC) and the CAN reception error counter (REC).

Interrupt status and control registers. When a CAN event occurs on a given CAN controller, an interrupt is triggered for that controller. The interrupt is known as the *CAN combined interrupt* of the controller and is the same for all CAN events that occur at that controller. The CPU of the microcontroller then accesses the interrupt vector table (IVT) and invokes the interrupt service routine (ISR) located at the interrupt vector corresponding to the CAN combined interrupt[1]. Afterwards, the invoked CAN combined ISR must discern what specific CAN event caused the CAN combined interrupt to be triggered. For this, each CAN controller has an interrupt flag register whose bits are flags that identify the specific CAN event that triggered the CAN combined interrupt. The interrupt flag registers are granular enough to discern among error warning; error passive; bus off; transmission error; reception error; transmission at transmission buffer 1, 2, or 3; reception at reception buffer 1 or 2; and other possible events. In addition, each CAN controller has a CAN interrupt enable register. With the bits of this register it is possible to specify what particular CAN events must trigger the CAN combined interrupt.

The above description is just a summary of the CAN controller registers. The complete details can be found in Section 23 of the dsPIC30F Family Reference Manual [Microchip, 2006b].

The large number of registers associated with the CAN controllers, and the even larger total number of control and status bit fields contained within those registers, makes the implementation of an ADT especially appropriate for the CAN controllers because in this way most of the complexity of dealing with the CAN registers is hidden from other parts of the driver program. Specifically, the CAN controller ADT abstracts away which specific control and status registers of the microcontroller—and which bits within those registers—need to be accessed for various control and status operations on the CAN controllers. Moreover, the CAN controller ADT provides the driver with additional status information and control operations specific to the media management the driver implements. For example, it allows the driver to know whether a specific CAN controller is a ReCANcentrate node's transmission controller or non-transmission controller, and whether a given CAN controller is active (that is, not quarantined) or not active (that is, quarantined).

Note, however, that because the driver is fairly low-level (close to the hardware), there are a few implementation details regarding the CAN controllers that we did not hide from other

[1] An *interrupt vector table* is a table in memory that contains a series of function addresses; the addresses contained within that table are known as *interrupt vectors* and the functions located at those addresses are known as *interrupt service routines*

parts of the driver. These implementation details are the fact that each CAN controller has two reception buffers and three transmission buffers, and that events occurring at these buffers generate interrupts. Nevertheless, the CAN controller ADT does not expose to other parts of the driver that each transmission and reception buffer is actually a collection of registers. For this purpose we implemented an additional ADT for the transmission buffers and another ADT for the reception buffers. Both of these ADTs are nested within the CAN controller ADT and their implementations are also found in the files can_controller.c and can_controller.h.

The LED ADT

The dsPICDEM board provides a row of four LEDs. An electronic circuit on the dsPICDEM board connects these LEDs to pins of the dsPIC30F6014A microcontroller [Microchip, 2006c]. These pins are pins of a general-purpose I/O port, called port D. Thus, to light the LEDs, the pins of port D that are connected to the LEDs must be configured as outputs. This is done through a so-called TRIS register, also known as a data direction register [Microchip, 2006b, Section 11]. In order to hide how the TRIS register of port D needs to be set up and what specific values then need to be transmitted through those pins to light the LEDs we implemented an ADT for the LEDs. The source code for this ADT can be found in the files led.c and led.h, which are listed in Section C.11 and C.12 of Appendix C respectively. The ADT is very simple. It provides one operation to initialize the LEDs, which basically sets up the TRIS register of port D, and another operation to display on the LEDs, in binary, a number passed as a parameter to the operation.

We mainly used the LEDs for debugging purposes: to alert us when an assertion failed, as described in Section 8.3.1, and, at times, as a simple substitute for *tracing statements*, that is, diagnostic messages that developers typically print to the screen in order to assess whether a particular piece of code gets executed or to determine the value of some variable.

The transmission timer ADT

The dsPIC30F6014A microcontroller has five embedded hardware timers. Of the five we used the first one, referred to as timer 1, as the transmission timer (see Section 5.3.2). We created a transmission timer ADT that provides three operations: an operation that enables the transmission timer so that it starts counting, an operation that disables it so that it stops counting, and an operation that resets its value to zero. By having these operations as part of an ADT, other parts of the driver source code did not have to concern themselves with the configuration register for timer 1. The source code for the transmission timer ADT is listed in sections C.19 and C.18 of Appendix C.

The interrupt ADT

The dsPIC30F6014A microcontroller has up to 41 interrupt sources [Microchip, 2006a, Section 5], all of which correspond to either hardware peripherals embedded in the microcontroller or to external interrupts, that is, interrupts that are generated by hardware peripherals external to the microcontroller.

There are no special-purpose instructions to generate software interrupts. At first, this might seem as a problem for us because the CAN event tracker needs to generate software interrupts in order to invoke the media management routines (see sections 5.3.3 and 5.3.4). But, luckily, all of the hardware interrupts can also be triggered by software; that is, an interrupt can be triggered either by a hardware peripheral or by the software writing a ‘1’ to the appropriate interrupt

flag. This means that we can reuse otherwise unused interrupt sources as software interrupts to trigger the media management routines. For instance, in our prototype we had no need for the first external interrupt, known as INT1; we therefore reused it as the software interrupt to trigger the ISR for the CAN1 rx routine. Thus, when the CAN event tracker wants to invoke the CAN1 rx routine, it simply sets the flag for the INT1 interrupt.

This works perfectly, but it is confusing to any developer reading the source code: a developer would ask why the INT1 interrupt triggers the rx routine. So, to make the source code more readable, we implemented an ADT for interrupts. The interrupt ADT is implemented in files interrupts.c and interrupts.h, listed in sections C.9 and C.10 of Appendix C respectively. Its main functionality is to provide an interface to the interrupts of the dsPIC30F6014A microcontroller that the driver needs. This interface abstracts away which specific interrupt enable, interrupt priority, and interrupt flag registers—and which bits of those registers—need to be accessed for the different tasks carried out by the driver. To accomplish this, the interrupt ADT provides a new `t_interrupt` enumerated data type and several operations on that new data type. The possible values for the new datatype are listed in the interrupts.h file. For easier reference, the relevant section of the file is reprinted here:

```
typedef enum t_interrupt_enum {
        /* Interrupts generated by hardware */
        HW_INTERRUPT_CAN1,
        HW_INTERRUPT_CAN2,
        HW_INTERRUPT_TIMER1,

        /* Interrupts generated through software by the
         * CAN event tracker */
        SW_INTERRUPT_CAN1_TX_EVENT,
        SW_INTERRUPT_CAN1_RX_EVENT,
        SW_INTERRUPT_CAN1_ERROR_WARNING,
        SW_INTERRUPT_CAN2_TX_EVENT,
        SW_INTERRUPT_CAN2_RX_EVENT,
        SW_INTERRUPT_CAN2_ERROR_WARNING
} t_interrupt;
```

The first three values for the `t_interrupt` datatype correspond respectively to the CAN1 combined interrupt, the CAN2 combined interrupt, and the interrupt of the transmission timer. The following six values represent software interrupts which are to be generated by the CAN event tracker. The interrupt ADT actually maps these software interrupts to unused hardware interrupt sources.

By ensuring that other parts of the driver source code only use interrupts through the interrupt ADT we increase the readability of the driver source code. First, the otherwise unused hardware interrupts that have been reused as software interrupts can be referred to by names that make semantically sense within the context of the driver. Second, the interrupt ADT provides a central point where an overview of the interrupts used by the driver is given. Additionally, the interrupt ADT makes the code more maintainable: if it later turns out that an interrupt that has been

recycled as a software interrupt is actually needed, only the interrupt ADT needs to be changed instead of every single line of the driver where that software interrupt had been used. Moreover, that change is very simple: it simply consists of a remap of the software interrupt to another unused hardware interrupt.

The interrupt ADT provides six operations: enabling interrupt nesting, setting the priority of interrupts, enabling a given interrupt, disabling a given interrupt, invoking a given interrupt by setting the appropriate interrupt flag, and clearing the interrupt flag corresponding to a given interrupt. All operations, except the one enabling interrupt nesting, receive as a parameter a `t_interrupt` datatype.

8.3.3. The CAN event tracker and the media management routines

The source code for the CAN event tracker can be found in file tracker.c, which is listed in Section C.16 of Appendix C. The source code for the rx routine can be found in files rxroutine.c and rxroutine.h, listed in sections C.14 and C.15 respectively; the code for the tx routine is found in file txroutine.c, listed in Section C.17; and the code for the qua routine is found in the file quaroutine.c, which is listed in Section C.13.

The tasks carried out by the CAN event tracker and the media management routines have already been described in detail in chapters 5 and 6. So we are not going to repeat what the tracker and the media management routines do. Instead, we want to highlight how we solved the problem of code duplication (see DRY principle, Section 8.2).

Remember that the CAN event tracker and each of the media management routines is actually implemented as two ISRs, one for each of the two CAN controllers. That means that, for instance, the ISR of the CAN1 rx routine and the ISR of the CAN2 rx routine are nearly identical and only differ in the CAN controller they consider to be the one that notified the reception. The most straightforward way to implement the rx routine ISRs would perhaps been the following. First write the code for, let us say, the ISR for the CAN1 rx routine. Then, in order to have an ISR for the CAN2 rx routine, copy and paste that code and replace all instances of CAN1 with CAN2 in the copy. However, this creates a lot of code duplication and violates the DRY principle.

A much better solution is possible thanks to the CAN controller ADT. The main functionality of each media management routine is implemented in a dedicated helper function that takes as a parameter two CAN controllers: the CAN controller that notified the delivery event/error and the other controller. These parameters are pointers to instances of the CAN controller ADT. A given ISR for a media management routine then simply calls the appropriate helper function passing as a parameter either a pointer to the CAN1 controller or a pointer to the CAN2 controller. To illustrate this, consider the following section of code, which is a simplified version of the relevant section of the rxroutine.c file:

```
extern volatile struct can_controller ctrl1, ctrl2;

/*
 * Handles a receive event notified by this_ctrl.
 */
```

```
static inline void handle_receive_event(
        volatile struct can_controller *const this_ctrl,
        volatile struct can_controller *const other_ctrl
)
{
        /* Implements the rx routine */
}


void __attribute__((__interrupt__, no_auto_psv))
_CAN1RxEventInterrupt(void)
{
        /* ACK software interrupt */
        clear_interrupt_flag(SW_INTERRUPT_CAN1_RX_EVENT);

        handle_receive_event(&ctrl1, &ctrl2);
}


void __attribute__((__interrupt__, no_auto_psv))
_CAN2RxEventInterrupt(void)
{
        /* ACK software interrupt */
        clear_interrupt_flag(SW_INTERRUPT_CAN2_RX_EVENT);

        handle_receive_event(&ctrl2, &ctrl1);
}
```

The function `_CAN1RxEventInterrupt()` is the ISR for the CAN1 reception event, the function `_CAN2RxEventInterrupt()` is the ISR for the CAN2 reception event, and the function `handle_receive_event()` is the rx routine helper function. Both ISRs start by acknowledging the corresponding software interrupt and then they simply call the rx routine helper function. The CAN1 reception ISR passes as parameters to the helper function first a pointer to the CAN1 controller and then a pointer to the CAN2 controller; whereas the CAN2 reception ISR passes the same parameters in reverse order to the helper function. This accomplishes what we want: when the helper function is called by the CAN1 reception ISR it implements an rx routine for the CAN1 controller and when the helper function is called by the CAN2 reception ISR it implements an rx routine for the CAN2 controller.

Moreover, note the `inline` keyword in the definition of the helper function. This means that when the appropriate optimization level is used during compilation, the body of the helper function will be appropriately inlined into the ISRs and there will be no actual call to the helper function. This removes the overhead of calling the helper function from the ISRs, resulting in an executable that should not be much slower than if we had used the copy and paste approach.

8.4. Implementation of a simple API to interface with the driver

We implemented a very simple application programming interface (API) for the ReCANcentrate driver that allows a user application executing on a node to transmit frames to other nodes and to receive frames from other nodes. The API implements the driver interface shown at the top of Figure 5.5, page 50. Moreover, as explained in Section 5.3.3, it abstracts away the existence of two CAN controllers, thereby allowing the user application to send and receive frames as if there was a single CAN controller. Note that this API is intended to be a proof of concept and has not been designed for real-world applications.

The source code for the API is found in the files recancentrate.c and recancentrate.h, which are both listed in Appendix D.

A user application that wants to use the driver must add an `include` statement of the recancentrate.h file and be linked with the driver object files to produce the final executable. Then it can use the functions declared in the recancentrate.h file. These functions are the following:

init_recancentrate_driver() The user application must invoke this function prior to any of the other functions provided by the API. It initializes the CAN controllers, and it enables the interrupts used by the driver and assigns them the adequate priorities .

request_recancentrate_tx() This function allows the user application to request the transmission of a frame through a ReCANcentrate network. It implements the tx request routine described in Chapter 6. The function takes four parameters. The first three parameters are input parameters and the fourth parameter is an output parameter. The input parameters are the identifier of the frame to be transmitted, the payload to be transmitted, and the length of the payload. The output parameter is used by the function to return a value that indicates whether the transmission request was successful or whether it failed because a transmission request was already pending. Note that the output parameter does not indicate whether the frame was actually transmitted successfully (for this see *recancentrate_tx_carried_out()* below), it only indicates whether the *request* to transmit was successful.

read_received_data() This function allows the user application to read a frame received from the ReCANcentrate network. It has four output parameters: the identifier of a received frame; the payload of a received frame; the length of the payload; and a status, which indicates whether or not there was a received frame and whether the other three parameters have been loaded.

received_data_is_available() This function returns true if the node has received a frame from another node. The user application can use this function to determine through polling when a frame has been received.

recancentrate_tx_carried_out() This function returns true if the last transmission request was carried out successfully and it returns false otherwise. The user application can use this function to determine through polling if the last frame that was requested for transmission has been successfully transmitted through the ReCANcentrate network.

recancentrate_controller_available() This function returns true as long as at least one of the two CAN controllers is still available for communication. It returns false when both CAN controllers have been quarantined and there are therefore no controllers left to communicate.

Using the above functions a node can communicate with other nodes on a ReCANcentrate network. As described above, a node uses polling to detect when a frame has been transmitted or when a frame has been received. We know that this is not very efficient and that it wastes processor cycles, but it was the easiest and fastest way to implement a simple API that is good enough for us. All that we needed was some means to write a few simple programs that would allow us to test the driver and the ReCANcentrate infrastructure that we built. If the API should ever be used by real-world applications, then it should probably be improved to allow a user application to write its own ISRs and to add them to the interrupt vector table. In this way the user defined ISRs would be invoked through interrupts as soon as a transmission request was fulfilled or a frame was received.

9. Testing the driver on the hardware prototype

This chapter describes the tests we carried out to verify the fault tolerance capabilities and performance of the new prototype. Note that these tests assumed that the hardware was working correctly because this had already been verified (see Section 7.4). Moreover, note that the focus of the tests described herein is on the nodes and not on the hubs, which had already been tested thoroughly in the prototype that Barranco et al. [2006a] built before this project. Specifically, the tests were intended to verify that the driver running on the nodes had been built according to the specification, that is, that the driver correctly implemented the flowcharts described in Chapter 6. But, more importantly, the tests were intended to validate if the driver and the node architecture are adequate to handle the replicated media provided by the ReCANcentrate hubs. Also, although we did not focus on testing the hubs, they did get exercised, which further increased our confidence in their correct functioning.

All tests were carried out using the network configuration of our prototype, that is, with two hubs, three nodes, and two interlinks. Moreover, we used the maximum achievable bit rate in all tests. Note that this was 921.25 Kbps, instead of CAN's maximum 1 Mbps, because with the 7.37 MHz oscillators of the nodes and the 14.74 MHz oscillators of the hubs we cannot reach more than 921.25 Kbps.

We begin this chapter by describing the tests we carried out to check whether the nodes of our prototype are capable of tolerating faults. Afterwards, the second part of this chapter deals with the performance tests of the driver.

9.1. Fault tolerance tests

Testing whether the nodes of our prototype are capable of tolerating faults required some means to inject faults. We therefore start this section by describing the implementation of fault injection.

9.1.1. Implementation of fault injection

In order to inject faults into the prototype, we designed and implemented, in VHDL, two different fault-injection modules for the hubs.

The first is the *downlink-fault-injection module*. As its name indicates, it allows to inject faults into a downlink of a hub. Figure 9.1 shows the internal structure of a ReCANcentrate hub (equivalent to Figure 5.3, page 45) with the additional new downlink-fault-injection module highlighted. The downlink-fault-injection module contains 3 units: a *fault-selection multiplexer* (MUX_{FS}), a *fault-enabling multiplexer* (MUX_{EF}), and a *fault-enabling unit*. Fault injection

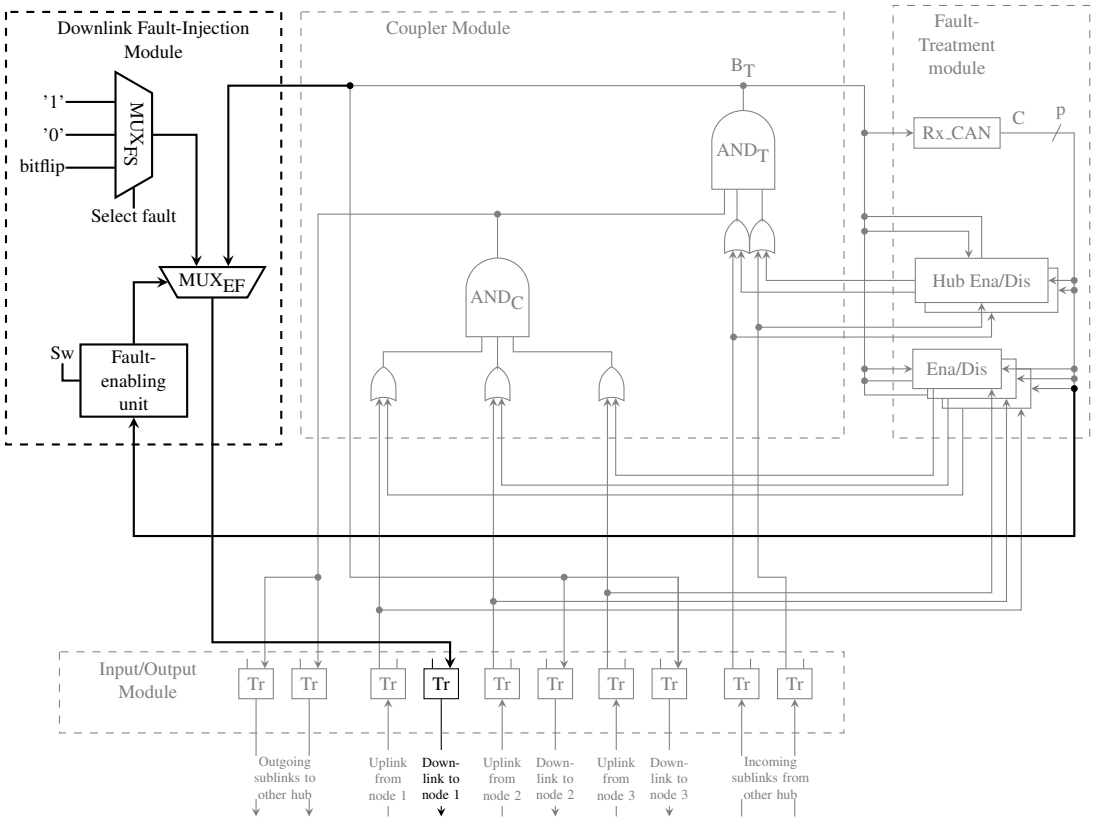

Figure 9.1.: Downlink-fault-injection module.

starts after the switch button (Sw) connected to the fault-enabling unit is pushed. Before the switch button is pushed, the fault-enabling multiplexer $\mathrm{MUX_{EF}}$ simply forwards the output $\mathrm{B_T}$ of the $\mathrm{AND_T}$ gate to the downlink of node 1. This is equivalent to what a ReCANcentrate hub without any fault-injection module does. Once the switch button has been pushed, the fault-enabling unit waits until it detects in the Rx_CAN's output that the CAN frame field at which fault injection should start is currently being broadcasted. Then it selects as output for the fault-enabling multiplexer $\mathrm{MUX_{EF}}$—and, thus, as input to node 1's downlink—the output of the fault-selection multiplexer $\mathrm{MUX_{FS}}$. The fault-selection multiplexer $\mathrm{MUX_{FS}}$ allows to choose the type of fault to inject: a stuck-at-recessive ('1'), a stuck-at-dominant ('0'), or a bit-flipping fault.

As an example consider the case where, once the switch button has been pressed, a stuck-at-dominant fault should be injected from the next data field being broadcasted onwards. For this, the downlink-fault-injection module must have been preconfigured appropriately by modifying its VHDL file. The preconfiguration is the following. First, the fault-selection multiplexer $\mathrm{MUX_{FS}}$ must select as an output the stuck-at-dominant fault. Second, the fault-enabling unit must change its output when the switch button has been pressed and the output of the Rx_CAN module indicates the broadcast of a data field. After this preconfiguration, the downlink to node 1

initially receives the coupled signal B_T. Then, when the switch button is pressed, the downlink to node 1 continues to receive the coupled signal B_T until the Rx_CAN module indicates that the coupled signal B_T corresponds to a data field. At that instant the fault-enabling unit is triggered to change its output. This causes the fault-enabling multiplexer MUX_{EF} to change its output: now it selects the stuck-at-dominant that is output by the fault-selection multiplexer MUX_{FS} instead of the coupled signal B_T. The result is that from that point in time onwards the downlink to node 1 receives a stuck-at-dominant bit stream.

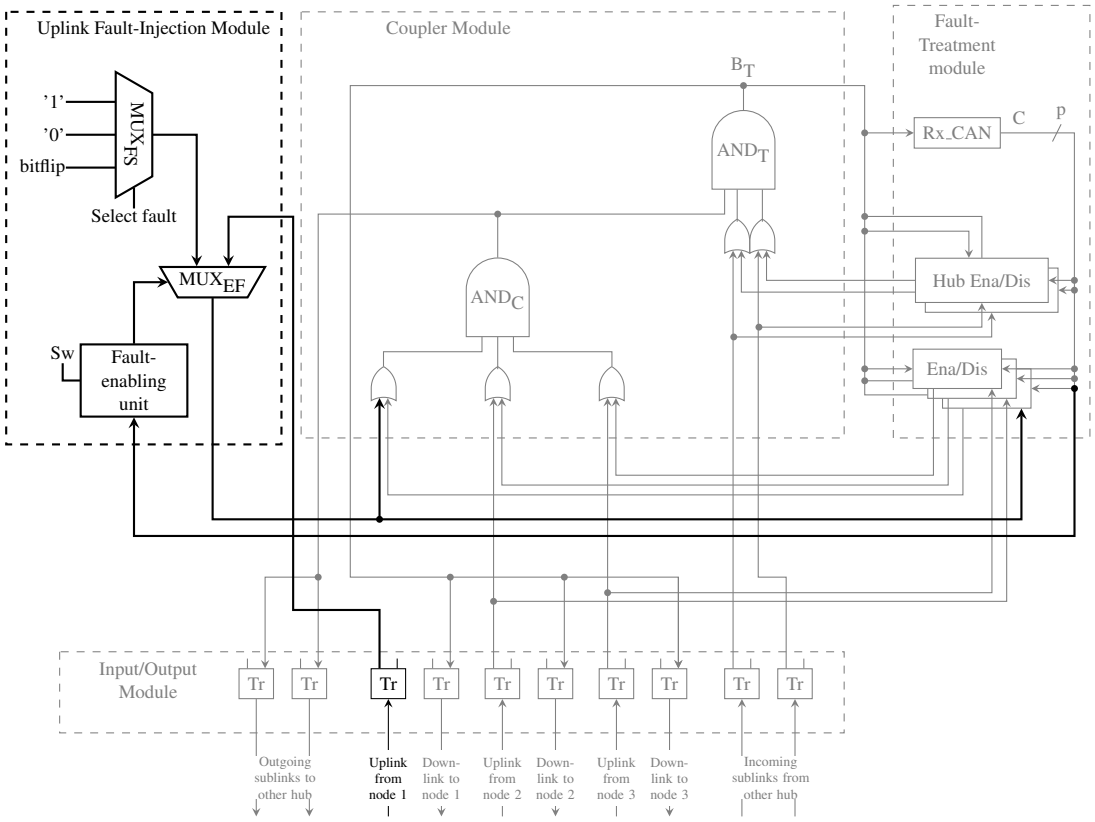

Figure 9.2.: Uplink-fault-injection module.

The second fault-injection module is the *uplink-fault-injection module*. It is shown in Figure 9.2. It is very similar to the downlink-fault-injection module shown in Figure 9.1: it also contains a fault-selection multiplexer (MUX_{FS}), a fault-enabling multiplexer (MUX_{EF}), and a fault-enabling unit. What changes with respect to the downlink-fault-injection module is where the right-most input to the fault-enabling multiplexer MUX_{EF} comes from and where its output goes to. Its right-most input is now the signal coming from node 1's uplink, and its output is now connected to both the OR gate and the enabling/disabling unit corresponding to node 1. As long as the switch button is not pressed, and thus no fault is injected, the result is equivalent to a ReCANcentrate hub with no fault-injection module: the signal from node 1's uplink enters both an OR gate and a corresponding enabling/disabling unit. But once a fault is injected, what enters

the OR gate and the corresponding enabling/disabling unit changes to the fault selected at the fault-selection multiplexer $\mathrm{MUX}_{\mathrm{FS}}$. The result is equivalent to a permanent fault, of the type selected, at the uplink of node 1.

The source code for the downlink-fault-injection module can be found in Section F.2.1 of Appendix F, and the source code for the uplink-fault-injection module can be found in Section F.2.2 of Appendix F. Both files implement the same interface, that is, seen as a black-box, they look the same from outside. However, inside one implements the downlink-fault-injection module and the other the uplink-fault-injection module. We modified the original VHDL code that we inherited from the previous prototype by Barranco et al.. Specifically, we changed the VHDL file that integrates and interconnects all the different modules of a ReCANcentrate hub in order to add the interface that is common to both fault-injection modules. Then, by synthesizing either one or the other implementation of the fault-injection module interface, we could program a given hub's FPGA to include either the downlink- or the uplink-fault-injection module. The file modified to integrate a fault-injection module can be found in Section F.2.3 of Appendix F.

Finally, apart from faults in the medium, the driver has also been designed to handle the crash of CAN controllers. To inject a controller crash we implemented a simple ISR for each controller that gets called when one of the push buttons on the dsPICDEM board is pushed. This ISR then disables all interrupts from the corresponding CAN controller and also shuts the controller down. To the driver this appeared as if the controller had suffered a crash and was no longer responding. The source code that allows the injection of CAN controller crashes can be found in Appendix F, Section F.1.

9.1.2. Test programs

The basis for all our fault-tolerance tests were 3 programs. The first program, named *3Bit-Counter* (listed in Appendix G, Section G.1), continuously increments an 8-bit counter, displays the value of that counter modulo 8 on the 3 right-most LEDs of the dsPICDEM board where 3BitCounter is running, and transmits the value of that counter in the payload of a 1-byte data frame. The second program, named *Blinker* (listed in Appendix G, Section G.2), also continuously increments an 8-bit counter and transmits the value of that counter in the payload of a 1-byte data frame. What is different is that it displays the value of that counter modulo 1 on the left-most LED of its dsPICDEM board. The third program, named *Receiver* (listed in Appendix G, Section G.3), continuously receives any frame transmitted by 3BitCounter or Blinker. It displays on its three least significant LEDs the counter values received from 3BitCounter modulo 8 and on its most significant LED the counter values received from Blinker modulo 1. The result on the 4 LEDs of Receiver is that the left-most LED blinks while the 3 right-most LEDs count in binary—although visually it seems that all 4 LEDs are continuously lit up, in less than the full brightness, because the LEDs switch their value too fast for the human eye. Moreover, using two local counters, Receiver keeps track of which was the last counter value it received from 3BitCounter and which was the last counter value it received from Blinker. It then compares the counter value received in each frame with the corresponding local counter. If the received counter value is synchronized with the corresponding local counter, Receiver determines that it correctly received the next frame from 3BitCounter or Blinker. Otherwise, it determines whether a single frame was missed, multiple frames were missed, or a duplicate of

the previous frame was received. Finally, Receiver includes assertions that check whether one or more frames were missed. If so, the assertions fail, thereby alerting us (through blinking LEDs, see Section 8.3.1) of this fact. Note that there are no assertions that fail if duplicate frames are received. This is so because the occurrence of duplicate frames is not considered an error—as described in Section 3.4.3, in CAN it is known that duplicates can occur and applications should be written so that they can deal with that.

In our tests we ran all 3 programs simultaneously, each one on a different node. The channel utilization was maximized, that is, the separation between each pair of consecutive frames was just the minimum intermission period (3 bits). We injected the faults described in the following sections and easily determined if the injected faults were tolerated: if Receiver had all of its 4 LEDs slightly lit up, it was receiving both 3BitCounter's and Blinker's frames. If a frame was missed, the corresponding assertion failed.

9.1.3. Test strategy

As we said in Section 5.3.2, nodes must deal with faults that manifest to them as syntactic faults in an up- or downlink, that is, they must deal with stuck-at, bit-flipping, and hub faults. Moreover, apart from syntactic faults, nodes must also be able to tolerate the crash of one of their CAN controllers and of certain CAN inconsistency scenarios. We have tested permanent syntactic faults and controller crashes, but we have not tested any scenarios that involve the CAN inconsistency scenarios that have been identified by Rufino et al. [1998] and Proenza and Miro-Julia [2000]. We have not tested these because injecting them would have required more sophisticated fault-injection mechanisms than we had available, and building those mechanisms was considered out of the scope of this project.

Regarding controller crashes, we did test a few scenarios that involve injected controller crashes. These are described in Section 9.1.8.

Regarding syntactic faults, there are four possible fault-injection points: the uplink of the transmission controller, the downlink of the transmission controller, the uplink of the non-transmission controller, and the downlink of the non-transmission controller. Each of these points can either be non-faulty, stuck-at-dominant, stuck-at-recessive, or bit flipping; that is, there are four possible fault states associated with each of the four injection points. There are therefore a total of $4 \times 4 \times 4 \times 4 = 256$ possible fault combinations to be considered for testing so far.

However, we do not only need to consider the type of faults that occur at a given fault-injection point, but for some types of faults we also need to consider *when* a given fault is injected. Although all faults in our fault model are permanent, the instant when a fault is injected can be relevant and can change how the node is expected to respond. This is true for stuck-at-recessive faults: when a permanent stuck-at-recessive occurs between the transmission of the start of frame (SOF) and the ACK delimiter of a frame, that frame will be corrupted and the affected controller will detect an error. However, when the stuck-at-recessive occurs anywhere between (and including) an ACK delimiter and the next SOF, the controller of the node affected by that stuck-at-recessive will not detect any error until it tries to transmit a frame. This is so because that controller will interpret the stuck-at-recessive stream as the last part of the frame and, afterwards, as an idle bus. It will continue with this interpretation until it monitors a recessive bit

when it tries to transmit an SOF.

We will refer to the period between an SOF and the next ACK delimiter as a *dominant including period* and to the period from an ACK delimiter to the next SOF as a *recessive only period*. Taking into account these periods for stuck-at-recessive faults, the number of possible fault states increases to five for each fault-injection point: a fault-injection point can be non-faulty, stuck-at-dominant, stuck-at-recessive that started during a dominant including period, stuck-at-recessive that started during a recessive only period, and bit flipping. This means that the possible fault combinations that in principle need to be tested increases to $5 \times 5 \times 5 \times 5 = 625$.

Unfortunately this is still not the number of all possible fault combinations that we need to consider for testing. Another thing that we need to take into account is the role (transmitter/receiver) of a node into whose link the fault is injected: if the node was only receiving frames and not transmitting any frames, then we do not expect any retransmissions to occur at that node. However, if the node was transmitting a frame when the permanent fault manifested, then in our tests we will also have to check if a retransmission of the corrupted frame occurs. Taking into account the role of a node doubles the number of fault scenarios to consider for testing, giving us $2 \times 625 = 1250$ fault scenarios.

Moreover, if multiple faults occur, then the delay between faults is also relevant. Consider the following example. A fault occurs at a link and is globalized through CAN's error signaling mechanism until it is isolated by the corresponding hub. This will increase the error counters (TEC/REC) of the controllers of all nodes—although, for the controllers that have a non-faulty link, they will not increase so far as to reach the error warning limit because the hub will have isolated the fault before that. After the first fault, the error counters of the controllers that have non-faulty links will start to decrease with each successfully transmitted or received frame. But if now a second fault occurs, the error counters of these controllers will start to increase again until that second fault is isolated by the corresponding hub. If that second fault occurs before the error counters had time to decrease enough, the error counters could reach the error warning limit. That means that, depending on the delay between the first fault and the second fault and the time it takes the hubs to isolate the fault, the increase of the error counters due to the second fault may cause the error counters of some controllers whose links are not faulty to reach the error warning limit. Thus, if we consider multiple faults, then, for certain fault scenarios, we need to take into account the delay between the faults in order to know if the nodes should quarantine their controllers in our tests or not. This further increases the number of fault scenarios to consider for our tests. Finally, even considering the delay between multiple faults, there are still more things that we need to consider in order to identify all fault scenarios which we may need to test. For instance, bit flipping faults can occur in many different patterns and with varying delays between successive bit flips.

From the above discussion it should be clear that the number of possible fault scenarios that we need to consider is very large—even infinite if we do not put some restrictions on the fault scenarios or group them into equivalent scenarios by some criteria. However, there is some hope. Not all considered fault scenarios need to be actually tested. Despite the fault-tolerance mechanisms of ReCANcentrate, many of the possible fault scenarios do not allow the affected node to communicate. For instance, there is no way a node can communicate if both its downlinks are stuck-at-dominant. To test these scenarios does not make any sense because we know that the node will not be able to communicate anyway. We can therefore restrict the number of tests to

those fault scenarios where we expect a node to tolerate the faults.

Moreover, there are many scenarios that can possibly be grouped and considered equivalent. In that case it may be enough to test a single scenario out of each group. But even with this strategy there are still too many scenarios to test them all manually, that is, with a person having to set up and supervise each test. If we wanted to test all scenarios exhaustively we would have needed some infrastructure for automated testing—which we did not have. So, for this project, we decided to test those scenarios where a single fault occurs with the additional restriction that the single fault can only be a stuck-at-recessive or stuck-at-dominant fault. We excluded bit-flipping faults as they would increase the number of tests to execute significantly: there are many different types of bit-flipping faults that need to be considered, depending on the bit-flipping pattern, the delay between successive bit flips, and where within a transmitted frame the bit flips coincide. Note, however, that many bit-flipping scenarios are equivalent to the injection of stuck-at-dominant faults. This is so because both bit-flipping and stuck-at-dominat faults corrupt frames.

We have identified 21 different groups of scenarios where a single permanent stuck-at-recessive or stuck-at-dominant fault occurs. For each of these 21 groups we have designed a test that represents the whole group. In other words, we have executed a total of 21 different representative tests involving either a single permanent stuck-at-recessive fault or a single permanent stuck-at-dominant fault. These tests are summarized in tables 9.1, 9.2, 9.3, and 9.4. We ran each test at least 10 times and, in all cases, observed that the injected fault was correctly isolated by the hub and tolerated by the node into whose downlink/uplink the fault was injected.

Nevertheless, despite all the tests having succeeded, we need to highlight that this unfortunately does not prove that further executions of our tests will also be tolerated by ReCANcentrate. Note that the circumstances between different executions of the same test may slightly vary because the setup of each test and the injection of a fault in each test required the intervention of a person. This means that the exact circumstances for a given test are not deterministic for different executions of that test—even though in each execution of the same test the nodes and the hubs have been configured identically. For this reason, a sample of only 10 executions for each test is not enough to convincingly show through experimentation, and from a statistical viewpoint, that ReCANcentrate tolerates the 21 groups of scenarios we identified. To have any confidence in the fault tolerance capabilities of ReCANcentrate, many more test executions would be required. This, however, would be very time intensive and tedious without an automated testing infrastructure (consider that the 10 executions for each of the 21 tests we executed means that we had to manually set up a test and inject a fault 210 times, and getting a statistically significant sample size of executions would require thousands of executions, which is completely unreasonable to do manually). Nevertheless, our tests did at least not falsify the hypothesis that ReCANcentrate can tolerate single permanent stuck-at-dominant and single permanent stuck-at-recessive faults, but instead supports that hypothesis.

9.1.4. Stuck-at-recessive downlink

Let us start by describing the tests in which we injected a stuck-at-recessive fault in a downlink during the recessive only period (see Section 9.1.3), which includes the ACK delimiter, the EOF, the interframe space, and the next SOF. During that period a receiving controller expects

Table 9.1.: Stuck-at recessive downlink-fault-injection tests

	Node	Injection point	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; `maxInconsists` retransmissions through new tx ctrl
7	Blinker	non-tx ctrl downlink	EOF	no global error; non-tx ctrl not quarantined (sees medium as idle); `maxInconsists` 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; `maxInconsists` retransmissions through tx ctrl

recessive bits anyway and, thus, the controller into whose downlink we injected the stuck-at-recessive fault will not detect any errors until it tries to transmit a frame itself. This leads to 3 possible subsequent scenarios:

(*a*) The stuck-at-recessive was injected into a downlink of a node that exclusively receives frames. The controller affected by the fault will simply see the medium as idle and, when another node transmits a frame, the affected controller will omit the notification of receptions without being quarantined, while the unaffected controller of the same node correctly receives the frames. (Covered by tests 1 and 2, Table 9.1.)

(*b*) The stuck-at-recessive was injected into the transmission controller's downlink of a node that, afterwards, initiates a transmission. The transmission controller will detect a bit error when it tries to transmit an SOF. It will therefore globalize the error by transmitting an error flag, which the other controllers will receive. The transmission controller itself, however, will not receive its own error flag nor any subsequent error flags transmitted by other controllers. It will therefore continue to detect further bit errors, which will also be globalized until the corresponding hub isolates the uplink through which the transmission controller keeps transmitting error flags. Once isolated, the transmission controller with the faulty downlink does no longer disturb the communication channel with its error flags. However, it itself continues to detect errors and, thus, its TEC keeps incrementing until it reaches the error warning limit. At that point the qua routine is invoked, which then quarantines the transmission controller, assigns the transmission controller role to the non-transmission controller, and instructs the new transmission controller to retransmit

the pending frame. The new transmission controller then retransmits `maxInconsists` times (see decision I of the tx routine flowchart, page 62) before the driver considers that the frame is successfully transmitted. Note that these retransmissions may be unnecessary, but they are a tradeoff that allows to tolerate in most cases the inconsistent message omission scenarios identified for CAN (see sections 3.4.3 and 6.4). Any further transmissions will be carried out by the new transmission controller. (Covered by test 5, Table 9.1.)

(*c*) The stuck-at-recessive was injected into the non-transmission controller's downlink of a node that, afterwards, initiates a transmission. The driver will observe that the transmission controller notifies the transmission, but the non-transmission controller omits the reception of the node's own frame. The transmission controller will retransmit because it must assume that the omission could have been due to a CAN inconsistency scenario and a retransmission is the correct thing to do in that case. In fact, since the non-transmission controller remains active but will permanently see the medium as idle, the transmission controller retransmits `maxInconsists` times (see decision I of the tx routine flowchart, page 62) before the driver considers that the frame is successfully transmitted. As in the previous case, these potentially unnecessary retransmissions are a tradeoff that allows to tolerate in most cases the inconsistent message omission scenarios identified for CAN (see sections 3.4.3 and 6.4). (Covered by test 7, Table 9.1.)

Next let us consider permanent stuck-at-recessive faults in downlinks that start during the dominant expecting period, which ranges from (and including) the first bit of a frame's identifier to (and including) the ACK slot of that same frame. During that period dominant bits are expected by a receiving controller at some points within the frame according to different rules, such as the bit stuffing rule. A transmitting controller expects to receive each bit value it transmits (except the ACK slot). Thus, the controller into whose downlink we inject the stuck-at-recessive fault will definitely detect an error and globalize it. We distinguish the subsequent scenarios:

(*d*) The stuck-at-recessive was injected into the non-transmission controller's downlink of a node that exclusively receives frames. The consequences are the following. First, the non-transmission controller will detect a stuff, CRC, or form error. In response it will transmit an error flag, which will be received by the other controllers; the non-transmission controller itself, however, will not receive its own error flag and will therefore detect bit errors during the error flag transmission. As a consequence, it will transmit even more error flags. This will cause the hub on the other end of the non-transmission controller's uplink to diagnose the non-transmission controller as stuck-at-dominant and to isolate its uplink. Once isolated, all other controllers (including the transmission controller of the same node) can resume the communication. The non-transmission controller with the faulty downlink, however, will continue to detect errors until it hits the error warning limit, at which point it is quarantined by the qua routine. (Covered by test 4, Table 9.1.)

(*e*) The stuck-at-recessive was injected into the transmission controller's downlink of a node that exclusively receives frames. The transmission controller will detect a stuff, CRC, or form error. Because of this, it will transmit an error flag through its uplink, which, however, it will not receive through its faulty downlink. This causes it to detect bit errors

and to transmit further error flags through its uplink, which again causes the corresponding hub to diagnose the uplink as stuck-at-dominant and to isolate it. Once isolated, the transmission controller with the faulty downlink no longer blocks with its error flags the communication between the other controllers. It itself, however, continues to detect bit errors until it hits the error warning limit. At this point the qua routine is invoked, which then quarantines the transmission controller and assigns the transmission controller role to the non-transmission controller. (Covered by test 3, Table 9.1.)

(*f*) The stuck-at-recessive was injected into the non-transmission controller's downlink of the node that was transmitting the frame in which the fault was injected. The non-transmission controller will detect a stuff, CRC, or form error; globalize the error; detect bit errors during the error flag transmission, which will trigger the transmission of further error flags; the nonstop transmission of the error flags will cause the corresponding hub to isolate the uplink of the non-transmission controller; and, afterwards, when it hits the error warning limit, the non-transmission controller will be quarantined. No retransmission is instructed by the qua routine, but the tx routine executing for the transmission controller will retransmit `maxInconsists` times (this again is unnecessary, but a tradeoff to tolerate some inconsistent message omission scenarios). (Covered by test 8, Table 9.1.)

(*g*) The stuck-at-recessive was injected into the transmission controller's downlink of the node that was transmitting the frame in which the fault was injected. The transmission controller will detect bit errors, globalize them until the corresponding hub isolates the uplink, then it will reach the error warning limit, and be quarantined by the qua routine. The non-transmission controller will become the new transmission controller, and it will be instructed by the qua routine to retransmit the corrupted frame. The tx routine executing on the new transmission controller will retransmit `maxInconsists` times (this is once more unnecessary, but, as in the previous cases, a tradeoff to tolerate some inconsistent message omission scenarios). (Covered by test 6, Table 9.1.)

9.1.5. Stuck-at-recessive uplink

When injecting a stuck-at-recessive fault into the uplink of a node, there are the following possible scenarios:

(*h*) The permanent stuck-at-recessive fault was injected into an uplink of a receiving node. In that case, as long as the affected controller continues to exclusively receive and does not transmit any frames itself, no controller detects an error because the stuck-at-recessive cannot corrupt any transmitted frame. Moreover, the stuck-at-recessive at the uplink does not prevent the affected controller from receiving frames. It does, however, prevent the affected controller from transmitting an ACK during an ACK slot. Nevertheless, because we only test scenarios where a single fault occurs, there will always be another controller that is able to transmit an ACK. Thus, although the controller into whose uplink we injected a stuck-at-recessive fault will not receive the ACK it transmits, it will detect an indistinguishable ACK from another controller (either of the same node or from another

Table 9.2.: Stuck-at-recessive uplink-fault-injection tests

	Node	Injection point	When	Observed consequences and fault-tolerance actions
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	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
12	Blinker	tx ctrl uplink	EOF	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; `maxInconsists` retransmissions through new tx ctrl
13	Blinker	tx ctrl uplink	data field	temporarily 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; `maxInconsists` retransmissions through new tx ctrl

node). The controller will therefore continue to notify the reception of frames as if no fault had occurred. (Covered by tests 9 and 10, Table 9.2.)

(*i*) The permanent stuck-at-recessive fault was injected into the uplink of the non-transmission controller of a transmitting node. In this scenario the non-transmission controller will not be able to transmit anything. However, as it is not being used for transmissions and it can continue to receive frames through its downlink, it will not detect any errors (we assume that there is at least one other non-faulty node that transmits ACKs); thus, it will not be quarantined and it will continue to notify receptions. (Covered by test 11, Table 9.2.)

(*j*) The permanent stuck-at-recessive fault was injected into the uplink of the transmission controller of a transmitting node during the recessive only period. In that case the transmission controller with the faulty uplink will detect bit errors at some point when it tries to transmit a frame. Note, however, that the transmission controller must not necessarily detect an error with its next transmission attempt: its next transmission attempt could lose the arbitration against a frame transmitted by another node. Specifically, note that although the transmission controller with the faulty uplink will not be able to transmit an SOF, it will not detect a bit error because there will be an SOF transmitted by the other node. If the frame transmitted by the other node has a higher priority, then the transmission controller with the faulty uplink will assume that it had lost the arbitration instead of that its uplink was stuck-at-recessive and it would therefore not detect a bit error yet. Nevertheless, at some point the controller with the faulty uplink will be the only one that wants to transmit a frame, or it will be the one to transmit a frame with the highest priority, and then it will detect a bit error. Anyway, when the bit error finally occurs, the transmission controller with the faulty uplink will not be able to globalize the error (it will therefore

be the only one to detect the error) and its TEC will reach the error warning limit. Then the qua routine will be invoked, and it will deactivate the transmission controller and assign the transmission controller role to the non-transmission controller. Moreover, the qua routine will instruct the transmission of the pending frame through the new transmission controller and `maxInconsists` (unnecessary) retransmissions will occur. (Covered by test 12, Table 9.2.)

(*k*) The permanent stuck-at-recessive fault was injected into the uplink of the transmission controller of a transmitting node during the dominant including period. In that case the frame that the node was transmitting will be corrupted. As a consequence the receiving controllers of the other nodes and the non-transmission controller of the transmitting node will detect a stuff, CRC, or form error and will therefore start to transmit error flags. The transmission controller of the transmitting node, into whose uplink we injected the stuck-at-recessive, will continuously detect a bit error, but it will not be able to transmit an error flag. Note that this scenario actually includes several subscenarios. Depending where exactly within the transmitted frame the stuck-at-recessive was injected, the transmission controller with the faulty uplink could either detect an error before any of the other, receiving, controllers or they could all detect the error at the same time. For instance, if the fault was injected during the data field, the transmitting transmission controller would detect the stuck-at-recessive as soon as it transmitted a dominant bit (it would detect a bit error); whereas the other controllers may assume that the transmitting controller had transmitted a recessive bit. The transmitting controller would not be able to globalize the error it has detected because it's uplink is stuck-at-recessive. Nevertheless, at most five bit times later, all controllers will have detected an error because either a stuff, CRC, or form error would have occurred. Anyway, at the end the transmitting transmission controller will be quarantined by the qua routine, the non-transmission controller of the transmitting node will become the new transmission controller, the qua routine will instruct a retransmission through the new transmission controller, and `maxInconsists` (unnecessary) retransmissions will occur. (Covered by test 13, Table 9.2.)

9.1.6. Stuck-at-dominant downlink

We now turn our focus to the stuck-at-dominant tests. As opposed to stuck-at-recessive faults, a long (more than 5 bits) series of stuck-at-dominant bits will always cause a CAN controller to detect an error. Thus, for permanent stuck-at-dominant faults we do not distinguish when the fault starts, but only where the fault was injected and whether the node was receiving or transmitting. For the case where the fault-injection point is a downlink, we have the following scenarios:

(*l*) If the permanent stuck-at-dominant fault was injected into the downlink of the transmission controller of a receiving node, the consequences are the following. First, the faulty stuck-at-dominant downlink will cause the transmission controller with the faulty downlink to detect errors, to which it responds by transmitting error flags. These error flags globalize the local stuck-at-dominant fault, preventing the further communication between

Table 9.3.: Stuck-at dominant downlink-fault-injection tests

	Node	Injection point	When	Observed consequences and fault-tolerance actions
14	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
15	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
16	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; `maxInconsists` retransmissions through new tx ctrl
17	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; `maxInconsists` retransmissions through tx ctrl

any controllers until the corresponding hub isolates the uplink of the transmission controller with the faulty downlink. Once isolated, the other controllers can communicate again; whereas the transmission controller with the faulty downlink continues to detect errors. At some point the transmission controller reaches the error warning limit and the qua routine is invoked on the corresponding node. The qua routine then deactivates the transmission controller and it assigns the transmission controller role to the other controller, which becomes the new transmission controller. (Covered by test 14, Table 9.3.)

(*m*) If the permanent stuck-at-dominant fault was injected into a receiving node's non-transmission controller downlink, the consequences are basically the same as for scenario (*l*). The differences are that the consequences apply to the non-transmission controller instead of the transmission controller, and that there is no reassignment of the transmission controller role from one controller to the other. (Covered by test 15, Table 9.3.)

(*n*) If the permanent stuck-at-dominant fault was injected into a transmitting node's transmission controller downlink, the consequences are basically the same as for scenario (*g*). The difference is that in scenario (*g*) the transmission controller detects bit errors when it tries to transmit its error flag; whereas in this scenario it does not detect errors during the transmission of its error flag, but instead never detects an error delimiter. (Covered by test 16, Table 9.3.)

(*o*) If the permanent stuck-at-dominant fault was injected into a transmitting node's non-transmission controller downlink, the consequences are basically the same as for scenario (*f*). Moreover, similar to the difference between scenario (*n*) and scenario (*g*), the difference between this scenario and scenario (*f*) is that in this scenario the non-transmission controller never detects an error-delimiter instead of detecting bit errors. (Covered by test 17, Table 9.3.)

9.1.7. Stuck-at-dominant uplink

For the case where the fault-injection point of the stuck-at-dominant is an uplink, we have the following scenarios:

(p) If the permanent stuck-at-dominant fault was injected into a receiving node's uplink, the consequences are the following. First, the dominant bits of the stuck-at-dominant will be received through the downlinks of all controllers, of all nodes, because of the single broadcast domain provided by the hubs. As a consequence, all controllers will respond by transmitting error flags until the faulty uplink is isolated by the corresponding hub. Note that the threshold of the hubs to isolate a port is lower than the error warning limit; thus, the faulty uplink is isolated before any of the controllers reaches the error warning limit. Once isolated, all controllers resume communication. The one with the faulty uplink will also communicate as long as it does not try to transmit a frame. This is so because the faulty uplink does not impede it from receiving frames and no bit error is detected during the ACK slot because it receives the ACKs sent by other controllers (remember that because we only consider single faults there will always be another controller transmitting ACKs). (Covered by tests 18 and 19, Table 9.4.)

(q) If the permanent stuck-at-dominant was injected into the uplink of the transmission controller of a transmitting node, the following occurs. As in the previous case, the dominant bits of the stuck-at-dominant will be received through the downlinks of all controllers, of all nodes, and, as a consequence, all controllers transmit error flags until the faulty uplink is isolated. Once isolated, the other controllers resume communication; whereas the one with the faulty uplink continues to detect errors because it tries to transmit frames through the isolated uplink. At some point it reaches the error warning limit, which invokes the qua routine. The qua routine then quarantines the controller, assigns the transmission controller role to the non-transmission controller, instructs the transmission of any pending frame through the new transmission controller, and `maxInconsists` (unnecessary) retransmissions occur. (Covered by test 20, Table 9.4.)

(r) If the permanent stuck-at-dominant was injected into the uplink of the non-transmission controller of a transmitting node, the consequences are basically the same as for scenario (p). Note that this means that, once the faulty uplink is isolated, the transmission controller of the transmitting node can continue to transmit frames. Also note that no retransmissions are instructed because the non-transmission controller continues to receive the frames transmitted by the transmission controller (assuming that some other controller is transmitting ACKs). (Covered by test 21, Table 9.4.)

9.1.8. Tests that inject controller crashes

We have carried out a series of tests to check whether the driver correctly handles the crash of a controller. As described in Section 9.1.1, the crash of a controller was injected through the manual push of a button on the dsPICDEM board. We tested the handling of controller crashes by the

Table 9.4.: Stuck-at-dominant uplink-fault-injection tests

	Node	Injection point	When	Observed consequences and fault-tolerance actions
18	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
19	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
20	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; `maxInconsists` retransmissions through new tx ctrl
21	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); both controllers continue to receive; tx ctrl continues to transmit

driver when the affected node was a receiving node and when it was a transmitting node. Moreover, we tested both the crash of a non-transmission controller and a transmission controller. The timing of the simulated crashes was arbitrary: they occurred shortly after the corresponding button on the dsPICDEM board was pressed. It may have been possible to precisely time the crashes; however, this would have required changes in the source code which could impact the behavior of the driver. For instance, the routine that compels the controller to act as if it had crashed would have had to have a higher priority than the driver's routines in order to inject the controller crash while the driver was executing; this would have affected the performance of the driver, which is crucial for its correct functioning as described in the next section.

In all controller-crash tests we verified, through the assertions in the driver's source code and by inspecting the oscilloscope (which probed the up- and downlinks of the affected node), that the crash was handled correctly.

9.2. Performance tests

The performance of the driver is important for its correct functioning since the driver must handle each delivery event and reset the respective tracking variables before a new delivery event occurs. If the performance is not good enough, any of the routines that must handle a given delivery event may not have finished before the next delivery event takes place. This could lead to a situation in which a media management routine corresponding to a new delivery event incorrectly cooperates with a media management routine corresponding to a previous one.

In all our fault tolerance tests (see Section 9.1) we maximized the channel utilization at 921.25 Kbps by having two transmitters: 3BitCounter and Blinker. In this way, whenever one of

the two just transmitted a frame, the other was already ready to transmit its own frame. Thus, the delay between successive frames was always the minimum. During these tests we checked, by means of the corresponding assertions in the code, that all delivery events were timely managed.

In addition to the fault tolerance tests, we also simultaneously ran the three test programs—3BitCounter, Blinker, and Receiver—during sessions of more than 8 hours in which no faults were injected. As in the fault tolerance tests, in these additional tests the channel utilization was also maximized at 921.25 Kbps. Moreover, as before, we checked that all delivery events where timely managed using the corresponding assertions in the code.

Although promising, this in no way proves that the driver will in all instances timely manage all delivery events. To confidently claim that the performance of the driver is good enough, we need to carry out a *timing analysis* of the driver's routines, that is, we need to establish an upper bound for the *worst-case execution time* (WCET) and check that this upper bound does not exceed the minimum available time to manage a delivery event.

In this section we briefly introduce the two basic methods that can be used to establish the WCET: *static methods* and *measurement-based methods*. Then we describe why we chose a measurement-based method and how we performed the measurement of the execution time of our driver. Afterwards we describe the scenario that we estimated to be the worst-case in terms of performance. Finally, we conclude this section by estimating the execution time of this worst-case scenario.

9.2.1. Methods to establish the worst-case execution time (WCET)

Establishing the WCET for a program is not trivial. For the most general case it is even impossible, that is, there is no general algorithm to establish the WCET of an arbitrary program because it would allow one to solve the halting problem, which is known to be undecidable. In other words, if the WCET analysis algorithm gave you an infinite amount of time as the WCET for a given program, you would know that the program does not halt, thereby giving you a solution to the halting problem, which is impossible. Nevertheless, for a subcategory of programs it is possible to calculate or, at the least, estimate the WCET. Specifically, the WCET can be established for programs that are known to halt because, for instance, they do not use any recursion and have explicit bounds for all loop counts [Wilhelm et al., 2008].

The execution time of a program depends on the input data or its environment. If we knew the input data or the conditions of the environment in which the program takes the longest time to execute, it would in principle be trivial to obtain the WCET: simply execute the program with that input or under the worst-case conditions and measure the execution time. Unfortunately, often the worst-case input and the worst-case conditions are not known and are not easy or even impossible to determine. Therefore, usually a timing analysis is done to obtain a value for the WCET [Wilhelm et al., 2008].

Timing analysis can be classified into two different types of methods: *static methods* and *measurement-based methods*. *Static methods* "do not rely on executing code on real hardware or on a simulator", but analyze the code of a program: they determine the possible control flow paths and, by using a model of the hardware architecture, derive upper bounds for the execution times of the different control flow paths. *Measurement-based methods*, on the other hand, execute the program or parts of it on the hardware or a simulator and obtain an estimate

for the WCET of the program from the measured execution times [Wilhelm et al., 2008].

Static methods have the advantage that they produce bounds instead of only estimates for the execution times. However, for the results to be precise, they need a detailed model of the hardware architecture on which the program is going to be executed—which is not that easy to obtain as modern processors contain caches, pipelines, branch prediction, and so forth. Measurement-based methods, in contrast, have the advantage that their results are more exact because they do not abstract the hardware architecture. However, they have the problem that it is generally not easy to know whether the longest observed execution time actually corresponds to the WCET. To be sure that the longest observed execution time does correspond to the WCET, we would have to measure all possible execution paths and each one of these paths would have to be measured under the worst-case conditions (with the caches set up so that they produce the maximum cache-miss penalties, with an execution history that disturbs the most the processor's pipeline, and so forth) [Wilhelm et al., 2008].

9.2.2. Performance measurement rationale

We have opted to use a measurement-based timing analysis approach instead of a static method. The three main reasons that led us to choose this option are the following:

- We did not have any tools available to support the static analysis of our code. This means that we would have had to build our own tools or that the analysis would have had to be done manually. Building our own tools would have been a whole project of its own. Doing the analysis manually, on the other hand, is tedious and error-prone; moreover, whenever we changed the source code or a compiler option, we would have had to redo this tedious and error-prone task. In contrast, a measurement-based approach is much quicker, less tedious, and the measurements can be somewhat automated.
- Although the CPU of the dsPIC30F6014A microcontroller is fairly simple (most instructions are single cycle and there are no caches, no out-of-order execution, and no branch prediction), the CPU does use pipelining. This means that data dependencies between instructions must be taken into account, which may lead to instruction stalls and makes the static analysis more complicated.
- We believe to have found the worst-case scenario in terms of performance—although we can obviously not be sure unless we rigorously explore the state space of all possible executions. So, to get the execution time for what we believe to be the worst-case scenario, we only need to set up that scenario and measure it in a test. We further explain this in the next subsection.

There are several ways to measure the execution time of a program. The ones we considered for the driver are the following:

Use an I/O pin of the microcontroller. The idea would be to set an otherwise unused pin of the microcontroller to a high value at the beginning of a given code section and to set it to a low value at the end of that code section. By connecting the probe of an oscilloscope to

that pin we can then measure the time to execute that code section. More specifically, the oscilloscope will display a pulse that starts when the pin is set high and that ends when the pin is set low; the width of that pulse can then be measured in time units with the oscilloscope, giving the execution time of the code section. We considered this option to be too inconvenient: it requires manually connecting the probes to the pins, inspecting visually the oscilloscopes display, and adjusting manually the oscilloscopes knobs to measure the pulse width—none of which can be automated. Also, the number of code sections that can be measured in one step, that is, without having to re-execute an experiment, is limited by the number of oscilloscopes we have.

Use the MPLAB SIM simulator to measure execution time. The MPLAB SIM simulator displays the total number of instruction cycles simulated for the program it has loaded. Moreover, it has a stopwatch that increases with each simulated instruction cycle and that can be reset to zero at any given instant. The stopwatch is intended to be used to measure the execution time between two points in the code, which is precisely what we need. Nevertheless, we did not use it to evaluate the performance of our driver's code because we found it to be too limited: the stopwatch is only available in the simulator and not when the driver is executed on the physical nodes; there is only a single stopwatch, but we wanted to measure in one step the execution time of the different sections of a trace; and the stopwatch is not programmable, but it has to be reset manually (that is, by clicking with the mouse on the appropriate button of the simulator's graphical user interface) before each measurement.

Use a code profiler. A code profiler is a software tool used to analyze the performance of a program. They typically collect data of a running program such as how many times each function is called and what percentage of the total execution time is spent in each function. There are various different types of profilers depending on how they collect the data. There are for instance statistical profilers, which probe the program counter of the running program at regular intervals; instrumenting profilers, which add additional instructions to the program that is to be measured; and simulator-based profilers, which execute the unmodified program under an instruction set simulator [Wikipedia, 2010b]. For programs running on desktop computers we can usually choose among many code profilers (some popular ones are for instance gprof [Fenlason and Stallman] and Valgrind [Seward et al.]). Unfortunately, for embedded systems it can be harder to find a code profiler. In our case we were not able to find a code profiler which we could use for the driver. Nevertheless, we had the option to implement a simple one ourselves, which is what we finally did.

We first experimented with compiler assisted instrumentation, that is, instrumentation where the compiler adds at the beginning and end of each and every function a call to a measurement function we define. However, at the end we implemented a simple instrumenting profiler with manual instrumentation, that is, an instrumenting profiler that required us to manually add to the functions that we wanted to measure calls to the profiler's measurement functions. We opted for manual instrumentation because it gave us better control and less of a performance hit. Consider that we only needed to measure certain functions—we were for instance not interested in the performance of the functions of the user application, but only in the performance of the driver functions. Thus, to us the advantage of manual instrumentation over compiler assisted

instrumentation was that in the first we could restrict the measurement overhead to only those functions whose execution time interests us; whereas in compiler assisted instrumentation an overhead is added to each and every function.

Appendix I lists the contents of the two files we used to implement the profiler: profiler.c and profiler.h. These files simply provide functions to start and stop the remaining four timers of the dsPIC30F6014A microcontroller (the fifth timer was already used as the driver's transmission timer). Thus, to measure a given section of code, we simply had to choose one of the four timers, start the timer at the beginning of the section of code by calling the appropriate profiler function, stop the timer at the end of the section of code by calling another profiler function, and then (using the ICD2 in-circuit debugger) inspect the contents of the timer registers to view the number of instruction cycles that have been measured. With this profiler we were able to measure four sections of code simultaneously, one with each timer, while the driver was executing on the physical hardware. Moreover, the timers are started and stopped by software and not through manual intervention, which made taking the measurements less tedious. Note, however, that the profiler introduces a small overhead.

9.2.3. Estimated worst-case scenario in terms of performance

The worst-case scenario in terms of performance occurs when the most work needs to be done and the least amount of time is available for that work. We believe that this occurs in the scenario whose chronogram is shown in Figure 9.3. In this scenario the node needs to execute an 8-byte-data transmission request, the CAN1 event tracker twice, the CAN2 event tracker, the CAN1 rx routine for the reception of an 8-byte-data frame, and the CAN2 rx routine; and all of this needs to be done before the shortest CAN frame—starting just after the minimum, 3-bit, intermission period—is broadcast.

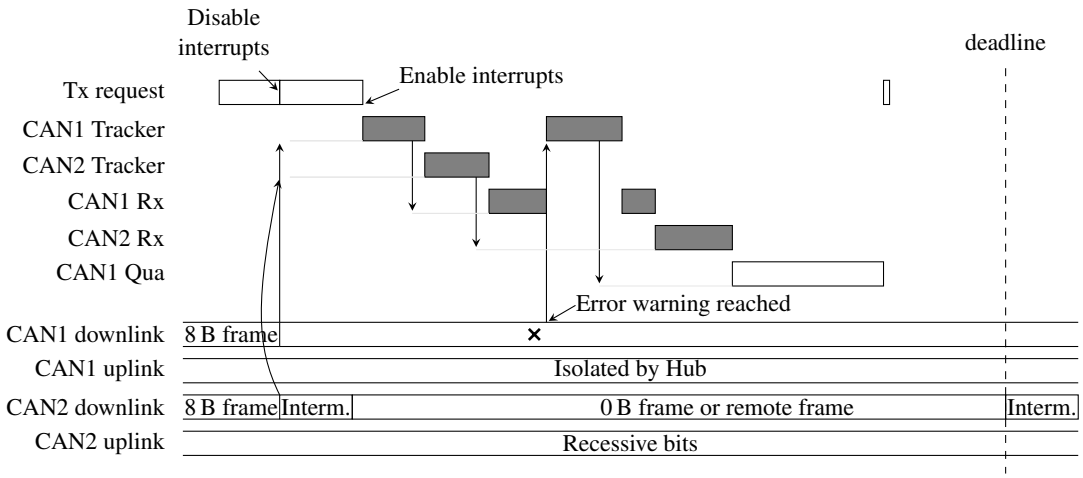

Figure 9.3.: Estimated worst-case scenario in terms of performance. The execution traces highlighted in gray must finish before the next delivery event, that is, before the shown deadline.

Let us discuss the chronogram in a little more detail. The chronogram begins with the recep-

tion of an 8-byte-data frame through the downlinks of the two controllers of the node. When the last EOF bit of that 8-byte-data frame is received, both controllers notify that reception by generating an interrupt (shown as upward arrows). This invokes the two CAN event tracker ISRs, one for each controller. However, the invocation is delayed because the reception of the frame coincided with a call to the tx request routine by the user application in such a way that the interrupts are disabled just when the frame is received. Moreover, the delay is the maximum possible because the transmission request was for the longest possible frame length, that is, 8 bytes—this means that the maximum amount of data has to be transferred to the transmission buffer of the transmission controller. After the delay, when the tx request routine enables the interrupts again, the pending CAN event tracker ISRs can start to execute. In the chronogram the first to execute is the one for the CAN1 controller. It determines that the cause for its invocation was a reception and, thus, generates a software interrupt (shown as a downward arrow) to invoke the ISR for the CAN1 rx routine. Next, the CAN2 event tracker ISR executes, which has a higher priority than the now also pending CAN1 rx routine ISR. It also determines that the cause for its invocation was a reception and, thus, generates a software interrupt to invoke the ISR for the CAN2 rx routine. After the CAN2 tracker ISR finishes, one of the pending rx routine ISRs executes. In the chronogram the one to execute first is the ISR for the CAN1 rx routine. It begins the management of the received frame, but, before it finishes, it is preempted by another invocation of the CAN1 tracker ISR. This second invocation occurs because of an error that only the CAN1 controller detects in its downlink and which made the REC of the CAN1 controller cross the error warning threshold. Note that the error is not globalized because the uplink of the CAN1 controller has been isolated due to prior errors (these are not shown in the chronogram). The second invocation of the CAN1 tracker ISR determines that it was invoked because the error warning limit was reached. It therefore generates a software interrupt to invoke the qua routine for the CAN1 controller. After it finishes, the previously interrupted CAN1 rx routine ISR resumes its execution. It concludes the management of the 8-byte-data frame and finishes. Afterwards, the CAN2 rx routine ISR executes—the also pending qua routine cannot execute yet because the implementation ensures that a qua routine is always executed after an also pending rx routine. The CAN2 rx routine ISR sees that the received frame has already been managed and simply finishes (see the flowchart of the rx routine, Figure 6.2, page 64). Finally, the qua routine starts to execute. In the scenario the CAN1 controller is the transmission controller. This means that the qua routine must not only deactivate the CAN1 controller, but also assign the transmission controller role to the CAN2 controller and, moreover, copy the frame whose transmission was requested at the beginning of the chronogram to the transmission buffer of the new transmission controller.

The execution traces that must finish before the next delivery event are highlighted in gray in the chronogram. The deadline, that is, the instant at which the next delivery event occurs, is shown as a dashed vertical line. This deadline is the strictest possible because it occurs after the shortest possible frame is exchanged following the minimum intermission. The qua routine is not highlighted: there is no need for it to finish before the next delivery event. That is so because the qua routine does not cooperate with any other media management routine but handles the deactivation of a controller on its own. In contrast, the other media management routines must finish timely because they could otherwise cooperate incorrectly with the media management routines of the next delivery event.

9.2.4. Performance measurement of the estimated worst-case scenario

We have measured the execution times of the traces of the primitives and ISRs involved in the worst-case scenario described above. Note, however, that the measurements were not taken by setting up a test that produces the worst-case scenario since our testing infrastructure was too limited to produce that scenario. Instead, we set up several tests (using the test programs from Appendix H) to produce parts of the worst-case scenario and we measured these parts. Afterwards, we combined these measurements to get the execution time of the worst-case scenario.

We measured the execution time of the driver for two different versions of the driver executable: one compiled with assertions and the other compiled without assertions. Apart from that, there was no difference between the two executables. For both executables the compiler used was the MPLAB C30 compiler, version 3.23, which is based on the GCC compiler, version 4.0.3. Both were compiled with the maximum optimization level (compiler option `-O3`), with function inlining enabled (`-finline`), and with the inline limit set to a sufficiently high value so that all code is inlined where ever possible (`--param max-inline-insns-single=900`). Note that this last compiler option increases the size of the executable.

Table 9.5 summarizes our measurements for the executable that includes assertions, and Figure 9.4 shows the chronogram of the worst-case scenario with these measurements plugged in. Note that in Figure 9.4, the execution traces are drawn to scale with respect to the bit time (32 instruction cycles correspond to each bit time). With assertions enabled the qua routine is delayed enough to interfere with the next delivery event. This is shown in the chronogram and is the reason why the table includes measurements for the second delivery event.

Specifically, what occurs when the qua routine interferes with the next delivery event is the following. When the last bit of the 0-byte-data frame or remote frame is received through the CAN2 downlink, the CAN2 controller notifies the reception—the CAN1 controller, on the other hand, does not notify the reception as it detected an error that prevented it from correctly receiving the frame. The notification from the CAN2 controller triggers the CAN2 tracker, which preempts the executing qua routine as the CAN2 tracker has a higher priority. The CAN2 tracker detects that it was invoked because of a notification by the CAN2 controller and, thus, generates a software interrupt to invoke the CAN2 rx routine. After the CAN2 tracker finishes, the qua routine resumes its execution, thereby delaying the start of the pending CAN2 rx routine[1]. Luckily, the amount of the delay is only 79 instruction cycles, which is a little more than 2 bit times. This is not enough to delay the rx routine so much that it will not have finished before the next, the third, delivery event (which is not shown in the chronogram) occurs.

[1] We said that when both a qua routine and an rx routine are pending, the qua routine must execute after the rx routine. In this case, however, the qua routine is not pending anymore but has already started its execution. Therefore, when the qua routine resumes its execution, it does not wait for the rx routine to finish first.

Table 9.5.: Measured execution time (in instruction cycles) for the media management routines (compiled with assertions) involved in the estimated WCET scenario.

	CAN1 Tracker (1st call)	CAN2 Tracker	CAN1 Tracker (2nd call)	CAN1 Rx	CAN2 Rx	CAN1 Qua
Delivery event 1:						
Interrupt latency	4	4	4	4	4	4
Save context	9	9	9	11	11	9
Start profiler	4	4	4	4	4	4
body + stop profiler	169	188	141	265	103	361
Restore context	9	9	9	12	12	9
Return	2	2	2	2	2	2
Total	197	216	169	298	136	389
Delivery event 2:						
Interrupt latency	—	4	—	—	4	—
Save context	—	9	—	—	11	—
Start profiler	—	4	—	—	4	—
body + stop profiler	—	188	—	—	205	—
Restore context	—	9	—	—	12	—
Return	—	2	—	—	2	—
Total	—	216	—	—	238	—

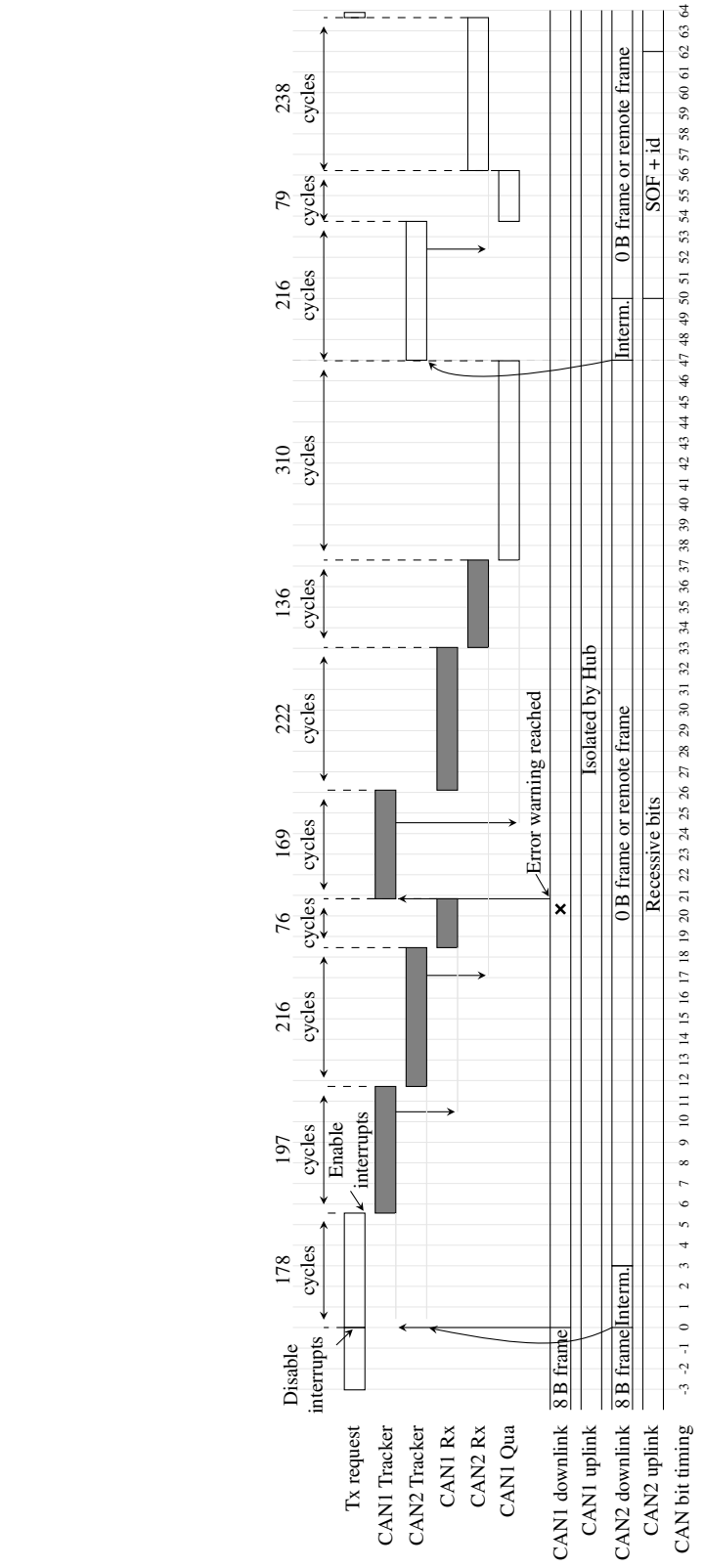

Figure 9.4.: Estimated worst-case scenario in terms of performance (with profiler overhead) compiled with assertions.

Table 9.6.: Measured execution time (in instruction cycles) for the media management routines (compiled without assertions) involved in the estimated WCET scenario.

	CAN1 Tracker (1st call)	CAN2 Tracker	CAN1 Tracker (2nd call)	CAN1 Rx	CAN2 Rx	CAN1 Qua
Interrupt latency	4	4	4	4	4	4
Save context	9	9	9	11	11	9
Start profiler	4	4	4	4	4	4
body + stop profiler	129	139	113	206	61	299
Restore context	9	9	9	12	12	9
Return	2	2	2	2	2	2
Total	157	167	141	239	94	327

Table 9.6 summarizes our measurements for the executable that has been compiled without assertions and Figure 9.5 shows the corresponding chronogram for the worst-case scenario. As can be seen in the chronogram, the performance improvement achieved by disabling the assertions is significant and the qua routine no longer interferes with the next delivery event.

The measurements that we have taken show that the driver performs timely in the scenario that we have estimated to be the worst-case. We want to make clear, however, that we have not found an upper bound for the worst-case execution time, but only an estimate for it. Nevertheless, we are very confident that the performance of the driver is not a problem. The reason for our confidence is that, even if it turns out that there is still a scenario that is even worse than our estimated worst-case scenario, we still have the possibility to improve the performance of the driver through code optimizations—remember that during the development of the driver we have always opted for readability, ease of maintenance, and modularity over performance (see Chapter 8). This means that there is still a fair amount of room for performance improvements on the source code level. Moreover, we also have the option to use a faster microcontroller.

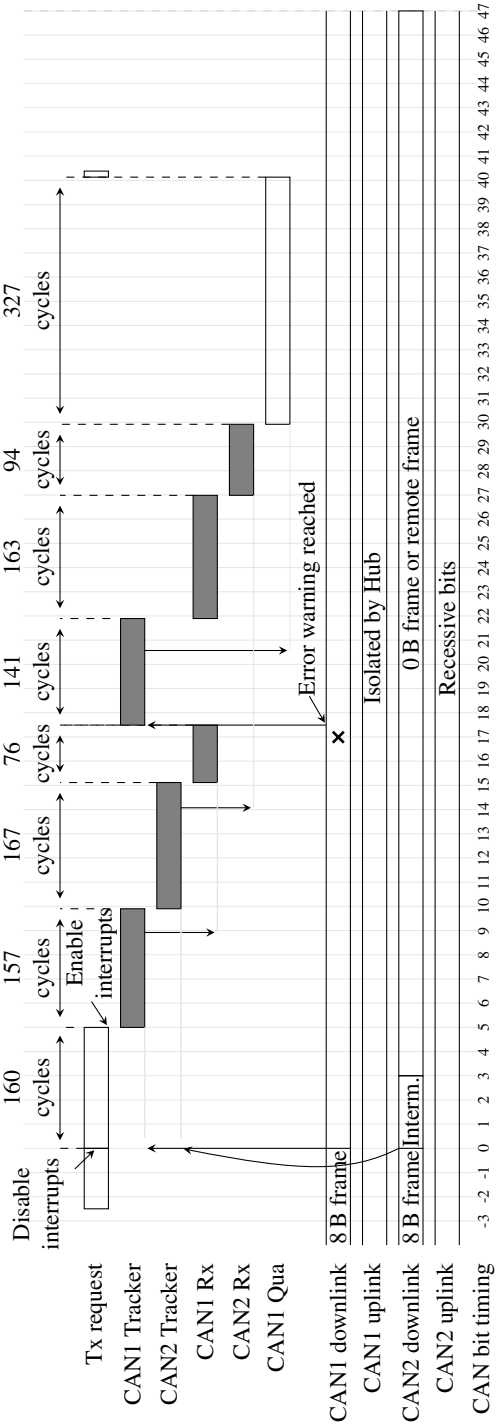

Figure 9.5.: Estimated worst-case scenario in terms of performance (with profiler overhead) compiled without assertions.

10. Conclusions

10.1. Summary

In this report we have presented the construction of a new hardware prototype of ReCANcentrate, which is based on a previous one that used simplified nodes, that is, nodes that do not implement all of the fault-tolerance mechanisms that have been designed for ReCANcentrate. In contrast to the nodes in the previous prototype, the nodes of the new one do implement all of these fault-tolerance mechanisms. Specifically, these mechanisms are implemented as a software driver for the nodes that allows applications running on these nodes to communicate through the ReCANcentrate star infrastructure. We have presented the design, implementation, and a first experimental validation of this software driver. We explained in detail the driver's media management routines and how they interact to handle receptions and transmissions both in the absence and in the presence of faults. We demonstrated that it is feasible to implement a ReCANcentrate network where the nodes implement all of the fault-tolerance mechanisms designed for them and that this can be done with commercial off-the-shelf (COTS) components. Moreover, we described the tests we executed in order to validate the fault-tolerance capabilities of the nodes and we explained why we had to compromise and did not exhaustively test all of the possible fault scenarios. We also carried out a measurement-based analysis of the worst-case execution time (WCET) of the driver. As with all measurement-based timing analysis methods, this did not give us an upper bound for the WCET, but it did give us an estimate for it. With this estimate we have demonstrated to some extent that the driver is always able to perform timely.

The project is a further contribution to the goal of demonstrating that the reliability of CAN networks can be improved while maintaining the key advantages of the CAN protocol: low cost, good real-time performance, and robustness to electromagnetically harsh environments.

10.2. Future work

Although some may have the opinion that a project to earn a degree as an *Ingeniero en Informática* should not leave work for the future, this project made it clear that there is future work to be done in order to convincingly show that ReCANcentrate tolerates all the faults of its fault model. It turned out that to exhaustively test our prototype is much more laborious than we initially anticipated. For this reason we plan to build a more sophisticated testing infrastructure and to use model checking techniques to formally verify the fault tolerance capabilities of the ReCANcentrate nodes.

The future testing infrastructure will hopefully allow us to show experimentally that the driver can not only deal with single stuck-at faults but also with bit-flipping faults, CAN inconsistency scenarios, and certain constellations of multiple faults. Moreover, we hope to collect in these

future tests enough experimental data for the tests to have a statistical significance.

Regarding our plans to use model checking, we should perhaps first briefly explain what model checking is. Model checking is a technique to formally verify a system. It consists in building a model of a system, typically as a set of automata, and then to formalize, using expressions in a specific logic, a set of properties that the model should hold. The model and the properties are then used as an input to a tool called a model checker. Based on the model, the tool then generates all the possible states the model can be in and verifies for each of these states that all the defined properties hold.

Manuel Barranco has already built a model of the ReCANcentrate media management driver, using the UPPAAL model checker [Behrmann, David, and Larsen, 2004], and verified that it holds the fault-tolerance properties that he formalized. This model, however, needs to be further refined as it does not take into account a few subtle things that we discovered during our experimental tests. This is actually one of the important lessons that we learned in this project: to demonstrate that a system satisfies certain requirements it is best to use both model checking and experimental tests. Model checking allows one to exhaustively test all the possible scenarios, but it depends on having a model that correctly reflects the real system; on the other hand, by executing experimental tests on the real system it is much harder to test all the possible scenarios, but it is very useful to detect subtleties that one may not have considered when initially modeling the system for model checking.

It has also remained pending to find a guaranteed upper bound for the WCET of the media management routines, which we need in order to demonstrate that the media management routines will timely manage a delivery event in all the scenarios in which the node should be able to communicate. For this we plan to explore all possible execution scenarios and assign an execution cost to each of them. We may do this using a hybrid timing analysis that combines static analysis with measurement-based analysis. More specifically, we may use a model checker to obtain the possible execution scenarios and afterwards we may use the measurements to assign a cost to these execution scenarios.

Apart from the tasks that still need to be carried out to convincingly demonstrate that ReCANcentrate tolerates all the faults of its fault model, there are other things related to this project that we still want to do, but that were not set out as goals for this project. For instance, the current design of the media management routines of the driver does not allow the reintegration of a CAN controller that has been quarantined if after some time the faults in the medium disappear. Currently, once a CAN controller has been quarantined, the driver does not use it again even if it turned out that the fault was not really permanent but just temporary. So, a further improvement to the driver may be to detect when a fault is no longer present and in that case rehabilitate the quarantined controller. Note that for this improvement we must carefully consider whether the once quarantined controller can be trusted again or whether it just appears to be no longer faulty.

Another improvement that is worth studying is the ability of a node to continue to communicate as long as one of its uplinks and one of its downlinks is correct, independently of which link they belong to—in the current implementation a link can only be used for communication if both the uplink and the downlink of that link are functioning correctly, that is, it is not possible for a controller to transmit through the uplink of one link and receive through the downlink of the other link. This improvement, however, cannot be implemented by solely modifying the driver, but would require additional hardware. It would therefore have to be studied if the ad-

ditional hardware would improve or reduce the reliability of the nodes because more hardware also means more possible components that could fail.

Moreover, a different approach worth studying is one where, instead of having a dedicated transmission controller, the driver alternates through which controller to transmit as long as both controllers and their links are correct. The advantage of this is that using each time a different controller to transmit allows the driver to better check the state of the redundancy and prevents an unnoticed redundancy attrition. In other words, the driver will detect sooner when a given controller is no longer able to transmit frames—although it may still be able to receive frames.

Finally, we propose a further improvement to the tx routine with the goal to avoid the unnecessary retransmissions that occur when a controller is quarantined (see sections 9.1.4, 9.1.5, 9.1.6, and 9.1.7). Although these retransmissions are not erroneous, they are unnecessary. We think that they can be avoided by simply checking whether the other controller is active before diagnosing a given delivery event as an inconsistent message omission. This would be achieved by adding to the tx routine's flowchart (see Figure 6.1, page 62) a new decision block, labeled K, as shown in Figure 10.1. Note that with this change the driver would have to be formally verified again and that the driver would have to be retested to ensure that the change does not produce unintended consequences.

10.3. Personal opinion

So far I have referred to myself in the first-person plural—the academic "we"—instead of in the first-person singular, because many readers consider the first-person singular to be very inappropriate for academic writing; here, however, I give my personal opinion and the first-person singular pronoun seems more appropriate.

I honestly enjoyed doing this project. I found it especially interesting as it involved work on many different levels: implementing electronic circuits and building the physical hardware using wirewrapping; studying, designing, and implementing digital logic circuits (in VHDL); gaining knowledge of a communication protocol (CAN) on its lowest levels; introducing myself into the field of fault tolerance, learning the design decisions (and trade-offs) involved in building a fault-tolerant system and how such a system can actually be implemented; helping to design the driver that I ultimately implemented for the ReCANcentrate nodes (during which I learned a few things about formal verification by means of model checking); doing a fair amount of embedded systems programming; testing on electronic circuit, logic circuit, and software level; learning about different timing analysis techniques and applying a particular one; getting introduced on how research is done; sharpening my English writing skills and my technical and academic writing skills (I chose English as it is the language in which research is communicated); and probably a few other things I have now forgotten.

Something that I personally find very noteworthy is that this project made it possible for me to get hired by the UIB as a technician and to assist in the further research of new ways to improve the reliability of CAN-based networks. This also means that the future work that is still to be done and which I described in the previous section will not be abandoned, but be part of my job duties.

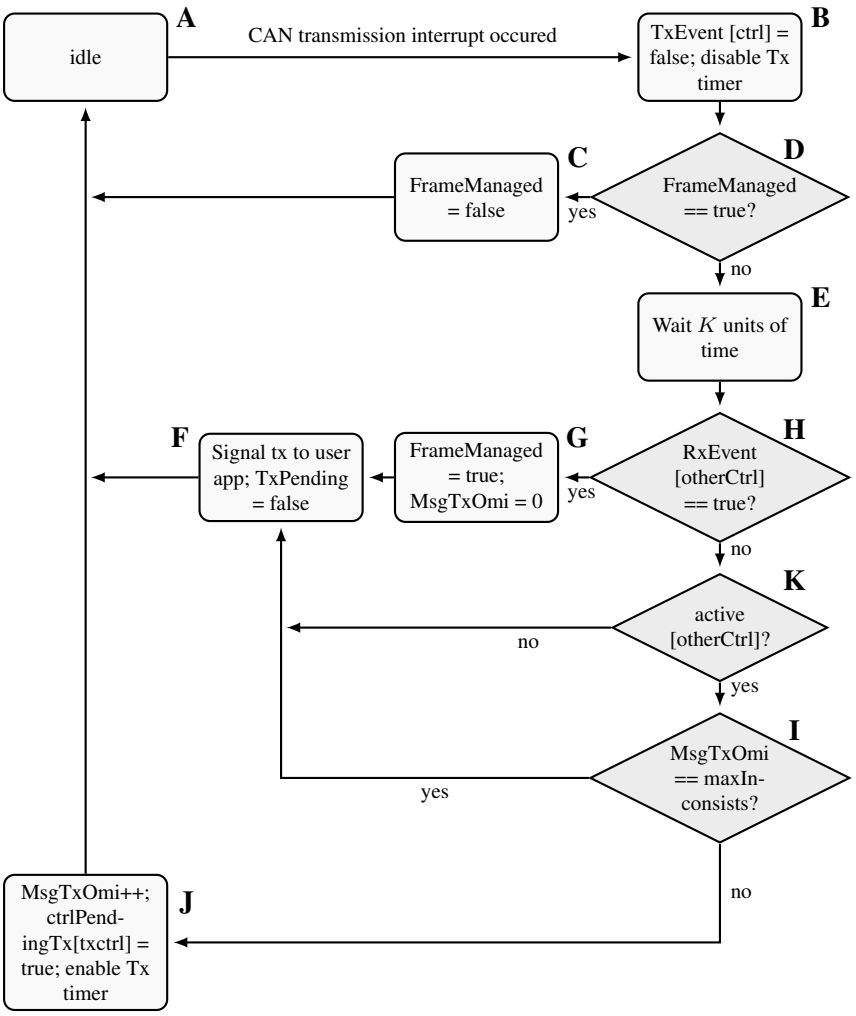

Figure 10.1.: Proposed improvement for the *tx routine*.

10.4. Publications

Finally, I want to conclude this report by highlighting three publications that I co-authored with Manuel Barranco, Julián Proenza, and Luís Almeida. All three publications are based on work carried out in this project. These publications are the following:

- Manuel Barranco, David Gessner, Julián Proenza, and Luís Almeida. Demonstrating the feasibility of media management in ReCANcentrate. In *14th IEEE international conference on Emerging Technologies and Factory Automation (ETFA), Mallorca, Spain*, September 2009.
- David Geßner, Manuel Barranco, Julián Proenza, and Luís Almeida. Evaluation of different approaches for the media management in ReCANcentrate nodes. Technical report A-01-2010, Universitat de les Illes Balears, July 2010.
- Manuel Barranco, David Geßner, Julián Proenza, and Luís Almeida. First prototype and experimental assessment of media management in ReCANcentrate. In *15th IEEE international conference on Emerging Technologies and Factory Automation (ETFA), Bilbao, Spain*, September 2010.

The first publication is a work-in-progress paper that presents the, back then, ongoing implementation of the media management driver. The second publication is a technical report that compares different approaches for the media management of ReCANcentrate nodes—among them the approach that we used in our prototype and the approach based on simplified nodes that was used in the prototype that was the precursor to the one we implemented for this project. The third publication is a full research paper that we presented in September 2010 at the ETFA conference. It describes the main features of the media management routines, of the driver implementation, and of the experimental stuck-at tests that we carried out.

Appendices

A. Initial design of the media management driver for ReCANcentrate

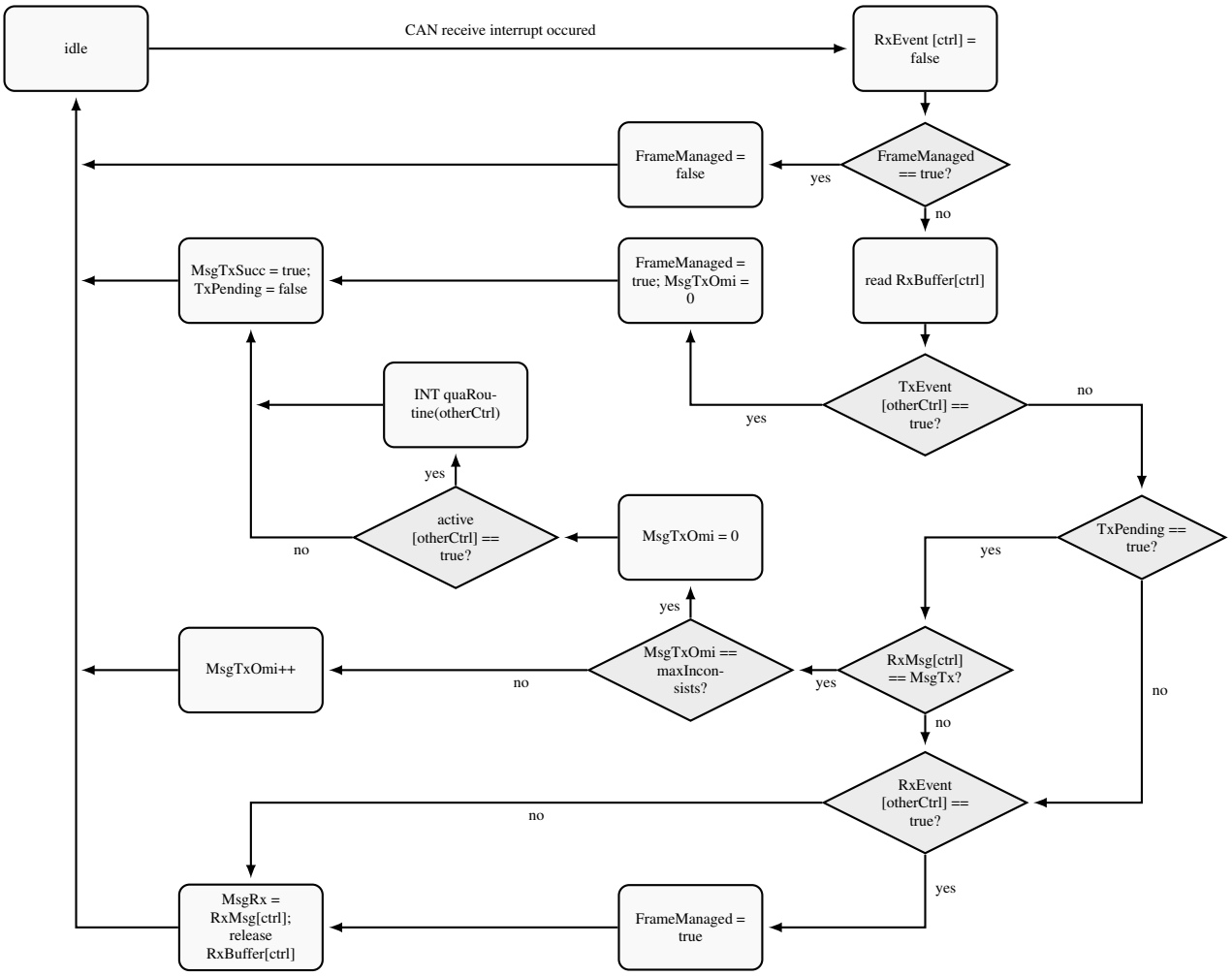

Figure A.1.: Initial design of the *rx routine*.

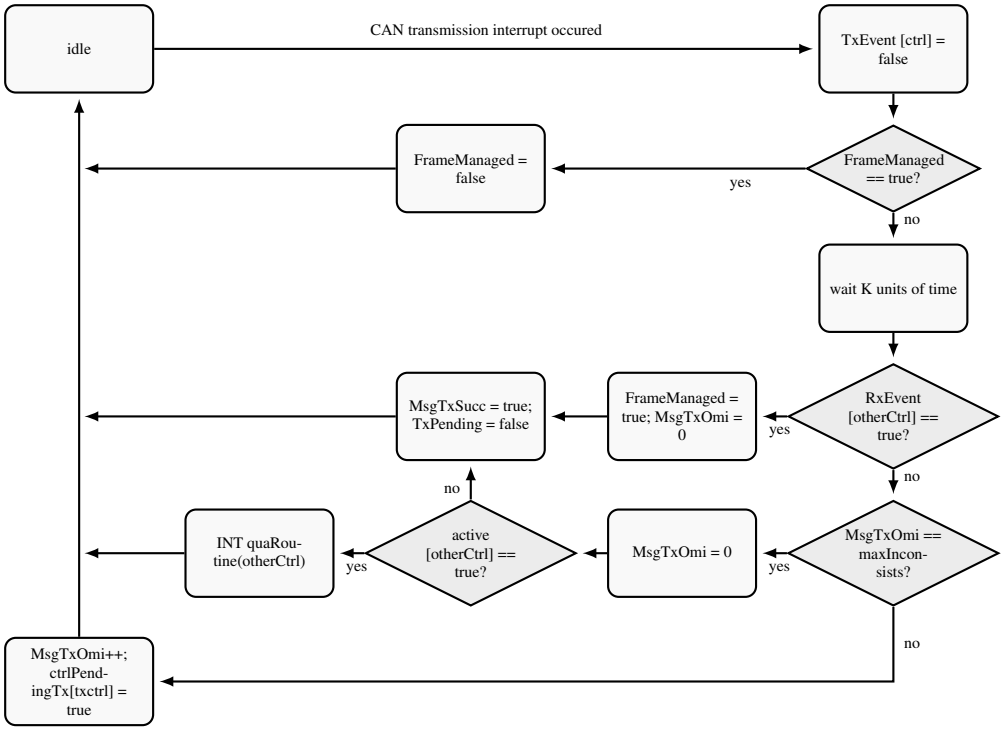

Figure A.2.: Initial design of the *tx routine*.

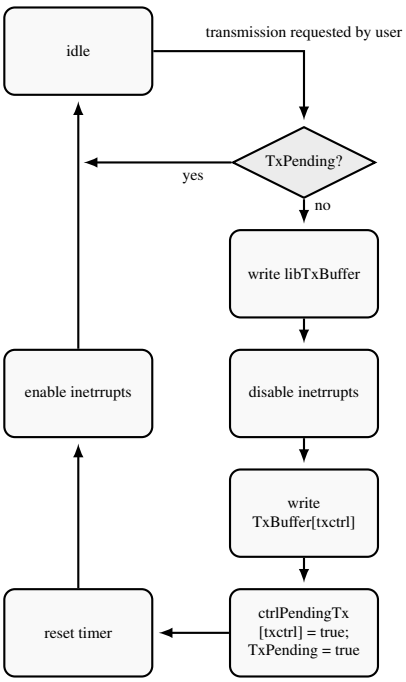

Figure A.3.: Initial design of the *tx request* routine.

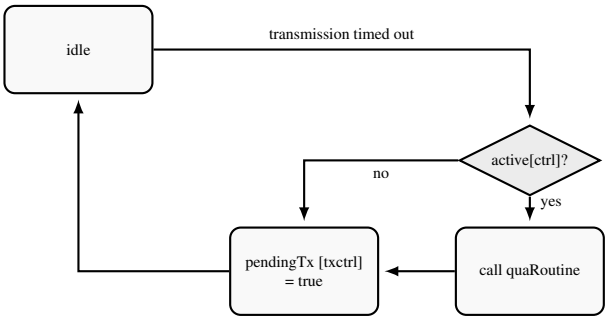

Figure A.4.: Initial design of the *tx timeout* routine.

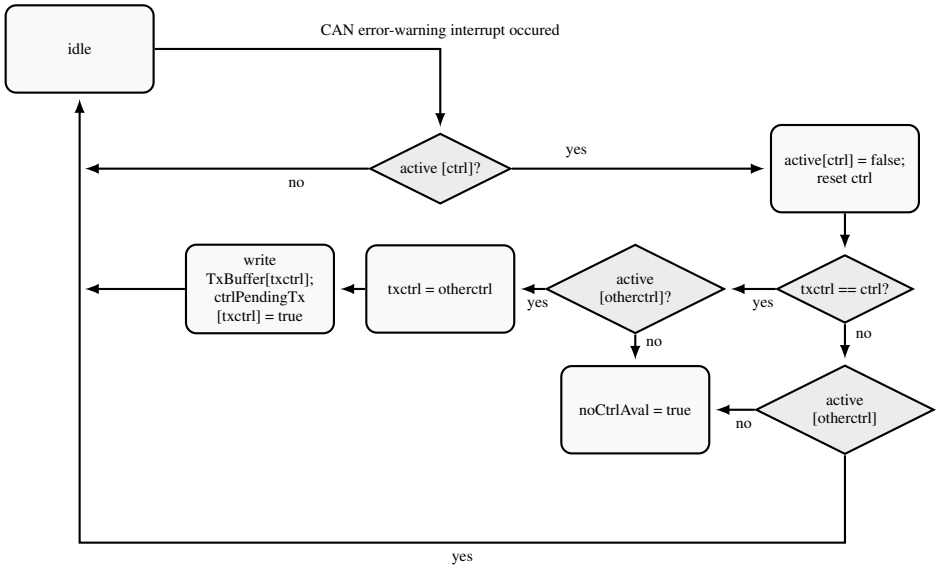

Figure A.5.: Initial design of the *qua routine*.

B. Source code for the preliminary tests

B.1. Header files used by the preliminary tests

B.1.1. can_aux.h

```
#ifndef __CAN_AUX_H_
#define __CAN_AUX_H_


/* CAN Module Operation Modes */
#define CAN_NORMAL_MODE            0
#define CAN_DISABLE_MODE           1
#define CAN_LOOPBACK_MODE          2
#define CAN_LISTEN_ONLY_MODE       3
#define CAN_CONFIG_MODE            4
#define CAN_LISTEN_ALL_MSGS_MODE   7


/* CAN receive buffers.
 * The CAN module has two visible receive buffers (the third buffer is
 * the message assembly buffer (MAB) and is not directly accessible) */
#define CAN_RXB0    0
#define CAN_RXB1    1

/* CAN transmit buffers */
#define CAN_TXB0    0
#define CAN_TXB1    1
#define CAN_TXB2    2

/*
 * CAN Baud Rate Prescaler.
 *
 * BRP_PLUS1 is the desired BRP value incremented by 1.
 *
 * Time quantum:
 *
 *          2 (BRP + 1)       2 (BRP_PLUS1)
 * TQ = ---------------- = ----------------
 *           FCAN              FCAN
 */
#define BRP_PLUS1       2   /* Valid values are 1, 2, ... 64 */


/*
 * Length in time quanta for each CAN bit time segment
 * (the SYNC segment is always 1 TQ)
```

```
 */
#define CAN_SEG1_TQ 5    /* Valid values are 1, 2, ... 8 */
#define CAN_SEG2_TQ 2    /* Valid values are 1, 2, ... 8 */
#define CAN_PROP_TQ 8    /* Valid values are 1, 2, ... 8 */

/* Restrictions on the CAN bit time segments (see the dsPIC30F Family Reference
 * Manual for details):
 *
 * CAN_PROP_TQ + CAN_SEG1_TQ >= CAN_SEG2_TQ
 * CAN_SEG2_TQ > Synchronous Jump Width
 *
 * The values for the Nominal Bit Time (NBT) must be between 8 * TQ and 25 * TQ.
 *
 * NBT = (1 + CAN_SEG1_TQ + CAN_SEG2_TQ + CAN_PROP_TQ) * TQ
 *
 * Therefore:
 * 8 <= (1 + CAN_SEG1_TQ + CAN_SEG2_TQ + CAN_PROP_TQ) <= 25
 */


#endif /* ___CAN_AUX_H_ */
```

B.1.2. device_config.h

```
#ifndef ___DEVICE_CONFIG_H_
#define ___DEVICE_CONFIG_H_

#include <p30F6014A.h>

/******** DEVICE CONFIGURATION ********/

// Oscillator configuration
_FOSC(CSW_FSCM_OFF &     /* Clock switching and fail safe clock monitor off, i.e.
                do not detect clock failures and do not switch over
                to internal FRC oscillator. */
    XT_PLL8);    /* Use crystall oscillator multiplying the clock speed
                by 8 with a Phase Locked-Loop. */

// Watchdog timer configuration
_FWDT(WDT_OFF); // Watchdog timer off.

// Reset configuration
_FBORPOR(PBOR_ON & BORV_20 &     /* Enable brown out at 20 volts. */
    PWRT_64 &    /* Power up timer = 64ms, gives the oscillator time to
                start and stabilize. */
    MCLR_EN);    /* Master clear reset enabled, i.e. use the MCLR pin
                as a reset signal instead of using it as an IO pin.
                Pulling the MCLR pin low will reset the dsPIC and
                start execution from 0x000. */

// General Code Segment configuration
_FGS(CODE_PROT_OFF);     // Disable Code Protection

#endif /* ___DEVICE_CONFIG_H_ */
```

B.1.3. dspicdem.h

```
#ifndef ___DSPICDEM_H_
#define ___DSPICDEM_H_

/*
 * dsPICDEM Starter Demo Board V2
 */

#define LED1      PORTDbits.RD4    // LED connected to RD4
#define LED2      PORTDbits.RD5    // LED connected to RD5
#define LED3      PORTDbits.RD6    // LED connected to RD6
#define LED4      PORTDbits.RD7    // LED connected to RD7

#define LED_ON  1
#define LED_OFF 0


#endif /* ___DSPICDEM_H_ */
```

B.1.4. portd.h

```
#ifndef ___PORTD_H_
#define ___PORTD_H_


void init_portd(void);


#endif /* ___PORTD_H_ */
```

B.2. Loopback test

B.2.1. canh_loopback.c

```
/*
 * CAN loopback demo for a dsPIC30F6014A on a dsPICDEM 80-Pin Starter
 * Development Board.
 *
 * This program sends a CAN frame in loopback mode. It uses polling to
 * determine if the frame was sent/received.
 */

#include <p30F6014A.h>
#include <can.h>
/* Required for can.h */
#define __dsPIC30F6014A__

#include "../../dspicdem.h"
#include "../../can_aux.h"
#include "../../device_config.h"
#include "../../portd.h"
```

```
/* Initialize CAN1 module */
void init_CAN1(void)
{
    int i;

    CAN1SetOperationMode(
        /* Stop CAN module when device enters idle mode. */
        CAN_IDLE_STOP &
        /* FCAN clock is FCY  */
        CAN_MASTERCLOCK_1 &
        /* Set configuration mode */
        CAN_REQ_OPERMODE_CONFIG &
        /* Don't generate a capture signal */
        CAN_CAPTURE_DIS
    );

    /* Wait until the CAN module has entered the configuration mode. */
    while (C1CTRLbits.OPMODE != CAN_CONFIG_MODE);

    CAN1Initialize(
        /* Synchronized jump width is 1 x TQ */
        CAN_SYNC_JUMP_WIDTH1 &
        /* BRP : TQ = 2x(BRP_PLUS1)/FCAN */
        CAN_BAUD_PRE_SCALE(BRP_PLUS1),
        /* CAN bus line filter is not used for wake-up */
        CAN_WAKEUP_BY_FILTER_DIS &
        /* Phase Segment 2 length is 3 x Tq */
        CAN_PHASE_SEG2_TQ(CAN_SEG2_TQ) &
        /* Phase Segment 1 length is 6 x Tq */
        CAN_PHASE_SEG1_TQ(CAN_SEG1_TQ) &
        /* Propagation Time Segment length is 5 x Tq */
        CAN_PROPAGATIONTIME_SEG_TQ(CAN_PROP_TQ) &
        /* The length of Phase Segment 2 is Freely programmable */
        CAN_SEG2_FREE_PROG &
        /* Bus line is sampled once at the sample point */
        CAN_SAMPLE1TIME
    );


    /* Configure receive buffers */
    for (i = CAN_RXB0; i <= CAN_RXB1; i++) {
        CAN1SetRXMode(i,
            /* Clear the receive full status register to indicate
             * that the receive register is empty and able to
             * accept a new message */
            CAN_RXFUL_CLEAR &
            /* Double buffer disabled, i.e.  if receive buffer 0 is
             * full don't overflow to receive buffer 1 */
            CAN_BUF0_DBLBUFFER_DIS
        );
    }
```

```
    /* Configure transmit buffers */
    for (i = CAN_TXB0; i <= CAN_TXB2; i++) {
        CAN1SetTXMode(i,
            /* Clears the transmit request bit, TXREQ. We don't
             * want to send a message yet */
            CAN_TX_STOP_REQ &
            /* Assign the same transmit priority to all three
             * buffers */
            CAN_TX_PRIORITY_HIGH
        );
    }

    /* Setup message acceptance filters and masks.
     *
     * The message acceptance filters and masks determine if a message in
     * the message assembly buffer (MAB) should be loaded into one of the
     * receive buffers. The filters and masks are applied to the message
     * identifier. The mask determines which bits of the identifier should
     * be examined and the filters contain values to which those bits are
     * compared. The bits from the identifier that are masked, i.e. the
     * corresponding mask bit is zero, will always be accepted by the
     * filters. The bits that are not masked, i.e. the corresponding mask
     * bit is one, will be accepted if there is a match with the
     * corresponding filter bit. If all the bits are accepted then the
     * message is accepted and loaded into one of the receive buffers.
     *
     * Messages whose identifier match filters RXF0 or RXF1 are loaded into
     * receive buffer 0 (RXB0), messages whose identifier match any of the
     * filters RXF2 through RXF5 are loaded into receive buffer 1 (RXB1).
     * The mask RXM0 is used with filters RXF0 and RXF1 and the mask RXM1
     * is used with filters RXF2–RXF5.
     */

#define MSG_SID 0x0AA8


    /* Make a filter that only matches an all zeros SID and EID */
    CAN1SetFilter(
        /* The filter to configure */
        0,
        /* The SID to match */
        CAN_FILTER_SID(MSG_SID) &
        /* Disable EID filtering (clears EXIDE bit). The filter will
         * accept standard identifiers (unless MIDE is cleared) */
        CAN_RX_EID_DIS,
        /* The EID to match, value doesn't matter as EID filtering is
         * disabled */
        0
    );


    /* Load mask filter register */
    CAN1SetMask(
        /* The mask to configure */
```

```
        0,
        /* Do not mask any SID bits */
        CAN_MASK_SID(0xFFFF) &
        /* Set MIDE bit. The EXIDE bit in filters RXF0 and RXF1 will
         * select between standard and extended identifiers */
        CAN_MATCH_FILTER_TYPE,
        /* Mask all EID bits */
        CAN_MASK_EID(0)
    );

    /* TODO: Should I initialize masks and filters I don't use???? */
}

int main(void)
{
    unsigned char data[1] = { 0xA };
    int datalen = 1;
    unsigned char datareceived[1];

    init_portd();

    init_CAN1();

    /* Set request for loopback mode */
    CAN1SetOperationMode(CAN_IDLE_STOP &
            CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_LOOPBK &
            CAN_CAPTURE_DIS);

    /* Load message ID and data into transmit buffer and set transmit
     * request bit */
    CAN1SendMessage(CAN_TX_SID(MSG_SID) &
            CAN_TX_EID_DIS &
            /* Send a normal (data) frame and not a remote
             * transmission request */
            CAN_SUB_NOR_TX_REQ,
            /* EID */
            0,
            data, datalen,
            /* Send with transmit buffer 0 */
            CAN_TXB0
    );


    /* Wait until the CAN module has entered the loopback mode */
    while (C1CTRLbits.OPMODE != CAN_LOOPBACK_MODE);

    LED1 = LED_ON;

    /* Wait until transmit buffer 0 has send the message */
    while (!CAN1IsTXReady(CAN_TXB0));

    LED2 = LED_ON;
```

```
    /* Wait until receive buffer zero contains a valid message */
    while (!CAN1IsRXReady(CAN_RXB0));

    LED3 = LED_ON;

    /* Read received data from receive buffer 0 and store it into user
     * defined dataarray */
    CAN1ReceiveMessage(datareceived, datalen, CAN_RXB0);

    if (datareceived[0] == data[0]) {
        LED4 = LED_ON;
    }

    while (1);

    return 0;
}
```

B.2.2. canh_loopbk_int.c

```
/*
 * CAN loopback demo for a dsPIC30F6014A on a dsPICDEM 80-Pin Starter
 * Development Board.
 *
 * This program sends a CAN frame in loopback mode. It uses interrupts to
 * determine if the frame was sent/received.
 */

#include <p30F6014A.h>
#include <can.h>
#define __dsPIC30F6014A__   /* Required for can.h */

#include "../../dspicdem.h"
#include "../../can_aux.h"
#include "../../device_config.h"
#include "../../portd.h"



/* Initialize CAN1 module */
void init_CAN1(void)
{
    int i;

    CAN1SetOperationMode(
        /* Stop CAN module when device enters idle mode. */
        CAN_IDLE_STOP &
        /* FCAN clock is FCY  */
        CAN_MASTERCLOCK_1 &
        /* Set configuration mode */
        CAN_REQ_OPERMODE_CONFIG &
        /* Don't generate a capture signal */
        CAN_CAPTURE_DIS
    );
```

```
/* Wait until the CAN module has entered the configuration mode. */
while (C1CTRLbits.OPMODE != CAN_CONFIG_MODE);

CAN1Initialize(
    /* Synchronized jump width is 1 x TQ */
    CAN_SYNC_JUMP_WIDTH1 &
    /* BRP : TQ = 2x(BRP_PLUS1)/FCAN */
    CAN_BAUD_PRE_SCALE(BRP_PLUS1),
    /* CAN bus line filter is not used for wake-up */
    CAN_WAKEUP_BY_FILTER_DIS &
    /* Phase Segment 2 length is 3 x Tq */
    CAN_PHASE_SEG2_TQ(CAN_SEG2_TQ) &
    /* Phase Segment 1 length is 6 x Tq */
    CAN_PHASE_SEG1_TQ(CAN_SEG1_TQ) &
    /* Propagation Time Segment length is 5 x Tq */
    CAN_PROPAGATIONTIME_SEG_TQ(CAN_PROP_TQ) &
    /* The length of Phase Segment 2 is Freely programmable */
    CAN_SEG2_FREE_PROG &
    /* Bus line is sampled once at the sample point */
    CAN_SAMPLE1TIME
);

ConfigIntCAN1(
    /* All interrupt sources are enabled */
    CAN_INDI_INVMESS_EN &
    CAN_INDI_WAK_EN &
    CAN_INDI_ERR_EN &
    CAN_INDI_TXB2_EN &
    CAN_INDI_TXB1_EN &
    CAN_INDI_TXB0_EN &
    CAN_INDI_RXB1_EN &
    CAN_INDI_RXB0_EN,

    CAN_INT_ENABLE &
    /* Interrupt priority is 1. 1 is the lowest priority, 7 is the
     * highest and 0 disables CAN as an interrupt source */
    CAN_INT_PRI_1
);


/* Enable interrupts */
EnableIntCAN1;

/* Configure receive buffers */
for (i = CAN_RXB0; i <= CAN_RXB1; i++) {
    CAN1SetRXMode(i,
        /* Clear the receive full status register to indicate
         * that the receive register is empty and able to
         * accept a new message */
        CAN_RXFUL_CLEAR &
        /* Double buffer disabled, i.e.  if receive buffer 0 is
         * full don't overflow to receive buffer 1 */
        CAN_BUF0_DBLBUFFER_DIS
```

```
        );
    }

    /* Configure transmit buffers */
    for (i = CAN_TXB0; i <= CAN_TXB2; i++) {
        CAN1SetTXMode(i,
            /* Clears the transmit request bit, TXREQ. We don't
             * want to send a message yet */
            CAN_TX_STOP_REQ &
            /* Assign the same transmit priority to all three
             * buffers */
            CAN_TX_PRIORITY_HIGH
        );
    }

    /* Setup message acceptance filters and masks.
     *
     * The message acceptance filters and masks determine if a message in
     * the message assembly buffer (MAB) should be loaded into one of the
     * receive buffers. The filters and masks are applied to the message
     * identifier. The mask determines which bits of the identifier should
     * be examined and the filters contain values to which those bits are
     * compared. The bits from the identifier that are masked, i.e. the
     * corresponding mask bit is zero, will always be accepted by the
     * filters. The bits that are not masked, i.e. the corresponding mask
     * bit is one, will be accepted if there is a match with the
     * corresponding filter bit. If all the bits are accepted then the
     * message is accepted and loaded into one of the receive buffers.
     *
     * Messages whose identifier match filters RXF0 or RXF1 are loaded into
     * receive buffer 0 (RXB0), messages whose identifier match any of the
     * filters RXF2 through RXF5 are loaded into receive buffer 1 (RXB1).
     * The mask RXM0 is used with filters RXF0 and RXF1 and the mask RXM1
     * is used with filters RXF2–RXF5.
     */

#define MSG_SID 0x0AA8


    /* Make a filter that only matches an all zeros SID and EID */
    CAN1SetFilter(
        /* The filter to configure */
        0,
        /* The SID to match */
        CAN_FILTER_SID(MSG_SID) &
        /* Disable EID filtering (clears EXIDE bit). The filter will
         * accept standard identifiers (unless MIDE is cleared) */
        CAN_RX_EID_DIS,
        /* The EID to match, value doesn't matter as EID filtering is
         * disabled */
        0
    );
```

```
    /* Load mask filter register */
    CAN1SetMask(
        /* The mask to configure */
        0,
        /* Do not mask any SID bits */
        CAN_MASK_SID(0xFFFF) &
        /* Set MIDE bit. The EXIDE bit in filters RXF0 and RXF1 will
         * select between standard and extended identifiers  */
        CAN_MATCH_FILTER_TYPE,
        /* Mask all EID bits */
        CAN_MASK_EID(0)
    );

    /* TODO: Should I initialize masks and filters I don't use????  */
}


unsigned char data[1] = { 0xA };
int datalen = 1;
unsigned char datareceived[1];

int main(void)
{
    init_portd();

    init_CAN1();

    /* Set request for loopback mode */
    CAN1SetOperationMode(CAN_IDLE_STOP &
            CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_LOOPBK &
            CAN_CAPTURE_DIS);

    /* Load message ID and data into transmit buffer and set transmit
     * request bit */
    CAN1SendMessage(
        CAN_TX_SID(MSG_SID) &
        CAN_TX_EID_DIS &
        /* Send a normal (data) frame and not a remote transmission
         * request */
        CAN_SUB_NOR_TX_REQ,
        /* EID */
        0,
        data, datalen,
        /* Send with transmit buffer 0 */
        CAN_TXB0
    );


    /* Wait until the CAN module has entered the loopback mode */
    while (C1CTRLbits.OPMODE != CAN_LOOPBACK_MODE);

    LED1 = LED_ON;
```

```
    /* Wait until transmit buffer 0 has send the message */
    while (!CAN1IsTXReady(CAN_TXB0));

    LED2 = LED_ON;


    while (1);


    return 0;

}


void __attribute__((interrupt, no_auto_psv)) _C1Interrupt(void)
{
    /* TODO: Should I call DisableIntCAN1? */

    /* TODO: What about nested interrupts? */

    /* Clear the CAN1 combined interrupt flag */
    IFS1bits.C1IF = 0;

    if (C1INTFbits.TX0IF) {
        /* The interrupt was due to a transmission through TXB0 of CAN1
         */
        /* Clear the interrupt */
        C1INTFbits.TX0IF = 0;
    } else if (C1INTFbits.TX1IF) {
        /* The interrupt was due to a transmission through TXB1 of CAN1
         */
        /* Clear the interrupt */
        C1INTFbits.TX1IF = 0;
    }

    if (C1INTFbits.RX0IF) {
        /* The interrupt was due to a reception at RXB0 of CAN1 */

        /* Clear the interrupt */
        C1INTFbits.RX0IF = 0;

        CAN1ReceiveMessage(datareceived, datalen, CAN_RXB0);

        LED3 = LED_ON;

        if (datareceived[0] == data[0]) {
            LED4 = LED_ON;
        }
    } else if (C1INTFbits.RX1IF) {
        /* The interrupt was due to a reception at RXB1 of CAN1 */

        /* Clear the interrupt */
        C1INTFbits.RX1IF = 0;
```

```
        CAN1ReceiveMessage(datareceived, datalen, CAN_RXB1);
    }
}
```

B.3. Single node test

B.3.1. one_node.c

```
/*
 * Sends a frame from CAN1 to CAN2. This program has been tested with one
 * dsPICDEM board connected to it's hardware input/output module. The hardware
 * input/output module must have the CANL/CANH of the transmit transceiver of
 * CAN1 connected to the CANL/CANH of the receive transceiver of CAN2. Also,
 * the CANL/CANH of the transmit receiver of CAN1 must be connected to the
 * CANL/CANH of the receive transceiver of CAN1.
 */

#include <p30F6014A.h>
#include <can.h>
/* Required for can.h */
#define __dsPIC30F6014A__

#include "../dspicdem.h"
#include "../can_aux.h"
#include "../device_config.h"
#include "../portd.h"


/* Initialize CAN1 module */
void init_CAN1(void)
{
    int i;

    CAN1SetOperationMode(
        /* Stop CAN module when device enters idle mode. */
        CAN_IDLE_STOP &
        /* FCAN clock is FCY  */
        CAN_MASTERCLOCK_1 &
        /* Set configuration mode */
        CAN_REQ_OPERMODE_CONFIG &
        /* Don't generate a capture signal */
        CAN_CAPTURE_DIS
    );

    /* Wait until the CAN module has entered the configuration mode. */
    while (C1CTRLbits.OPMODE != CAN_CONFIG_MODE);

    CAN1Initialize(
        /* Synchronized jump width is 1 x TQ */
        CAN_SYNC_JUMP_WIDTH1 &
        /* BRP : TQ = 2x(BRP_PLUS1)/FCAN */
        CAN_BAUD_PRE_SCALE(BRP_PLUS1),
        /* CAN bus line filter is not used for wake-up */
```

```
        CAN_WAKEUP_BY_FILTER_DIS &
        /* Phase Segment 2 length is 3 x Tq */
        CAN_PHASE_SEG2_TQ(CAN_SEG2_TQ) &
        /* Phase Segment 1 length is 6 x Tq */
        CAN_PHASE_SEG1_TQ(CAN_SEG1_TQ) &
        /* Propagation Time Segment length is 5 x Tq */
        CAN_PROPAGATIONTIME_SEG_TQ(CAN_PROP_TQ) &
        /* The length of Phase Segment 2 is Freely programmable */
        CAN_SEG2_FREE_PROG &
        /* Bus line is sampled once at the sample point */
        CAN_SAMPLE1TIME
    );


    /* Configure transmit buffers */
    for (i = CAN_TXB0; i <= CAN_TXB2; i++) {
        CAN1SetTXMode(i,
            /* Clears the transmit request bit, TXREQ. We don't
             * want to send a message yet */
            CAN_TX_STOP_REQ &
            /* Assign the same transmit priority to all three
             * buffers */
            CAN_TX_PRIORITY_HIGH
        );
    }
}

/* Initialize CAN2 module, the receiving module */
void init_CAN2(void)
{
    int i;

    CAN2SetOperationMode(
        /* Stop CAN module when device enters idle mode. */
        CAN_IDLE_STOP &
        /* FCAN clock is FCY  */
        CAN_MASTERCLOCK_1 &
        /* Set configuration mode */
        CAN_REQ_OPERMODE_CONFIG &
        /* Don't generate a capture signal */
        CAN_CAPTURE_DIS
    );

    /* Wait until the CAN module has entered the configuration mode. */
    while (C2CTRLbits.OPMODE != CAN_CONFIG_MODE);

    CAN2Initialize(
        /* Synchronized jump width is 1 x TQ */
        CAN_SYNC_JUMP_WIDTH1 &
        /* BRP : TQ = 2x(BRP_PLUS1)/FCAN */
        CAN_BAUD_PRE_SCALE(BRP_PLUS1),
        /* CAN bus line filter is not used for wake-up */
        CAN_WAKEUP_BY_FILTER_DIS &
        /* Phase Segment 2 length is 3 x Tq */
```

```
        CAN_PHASE_SEG2_TQ(CAN_SEG2_TQ) &
        /* Phase Segment 1 length is 6 x Tq */
        CAN_PHASE_SEG1_TQ(CAN_SEG1_TQ) &
        /* Propagation Time Segment length is 5 x Tq */
        CAN_PROPAGATIONTIME_SEG_TQ(CAN_PROP_TQ) &
        /* The length of Phase Segment 2 is Freely programmable */
        CAN_SEG2_FREE_PROG &
        /* Bus line is sampled once at the sample point */
        CAN_SAMPLE1TIME
    );


    /* The CAN module has two visible receive buffers (the third buffer is
     * the message assembly buffer (MAB) and is not directly accessible) */
    for (i = CAN_RXB0; i <= CAN_RXB1; i++) {
        /* Configure receive buffer i */
        CAN2SetRXMode(i,
            /* Clear the receive full status register to indicate
             * that the receive register is empty and able to
             * accept a new message */
            CAN_RXFUL_CLEAR &
            /* Double buffer disabled, i.e. if receive buffer 0 is
             * full don't overflow to receive buffer 1 */
            CAN_BUF0_DBLBUFFER_DIS
        );
    }


    /* Setup message acceptance filters and masks.
     *
     * The message acceptance filters and masks determine if a message in
     * the message assembly buffer (MAB) should be loaded into one of the
     * receive buffers. The filters and masks are applied to the message
     * identifier. The mask determines which bits of the identifier should
     * be examined and the filters contain values to which those bits are
     * compared. The bits from the identifier that are masked, i.e. the
     * corresponding mask bit is zero, will always be accepted by the
     * filters. The bits that are not masked, i.e. the corresponding mask
     * bit is one, will be accepted if there is a match with the
     * corresponding filter bit. If all the bits are accepted then the
     * message is accepted and loaded into one of the receive buffers.
     *
     * Messages whose identifier match filters RXF0 or RXF1 are loaded into
     * receive buffer 0 (RXB0), messages whose identifier match any of the
     * filters RXF2 through RXF5 are loaded into receive buffer 1 (RXB1).
     * The mask RXM0 is used with filters RXF0 and RXF1 and the mask RXM1
     * is used with filters RXF2–RXF5.
     *
     * Uninitialized filters and masks contain unknown values.
     */

#define MSG_SID 0x0AAF
```

```
    /* Setup both of RXB0's filters */
    for (i = 0; i < 2; i++) {
        CAN2SetFilter(
            /* The filter to configure */
            i,
            /* The SID to match */
            CAN_FILTER_SID(MSG_SID) &
            /* Disable EID filtering (clears EXIDE bit). The filter
             * will accept standard identifiers (unless MIDE is
             * cleared) */
            CAN_RX_EID_DIS,
            /* The EID to match, value doesn't matter as EID
             * filtering is disabled */
            0);
    }


    /* Load mask filter register */
    CAN2SetMask(
        /* The mask to configure */
        0,
        /* Do not mask any SID bits */
        CAN_MASK_SID(0xFFFF) &
        /* Set MIDE bit. The EXIDE bit in filters RXF0 and RXF1 will
         * select between standard and extended identifiers  */
        CAN_MATCH_FILTER_TYPE,
        /* Mask all EID bits */
        CAN_MASK_EID(0)
    );

    /* We do not configure masks and filters of RXB1 because we don't use
     * RXB1 */
}


int main(void)
{
    unsigned char data[1] = { 0x0A };
    int datalen = 1;
    unsigned char datareceived[1];

    init_portd();

    init_CAN1();
    init_CAN2();

    /* Set request for normal mode */
    CAN1SetOperationMode(CAN_IDLE_STOP &
            CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_NOR &
            CAN_CAPTURE_DIS);

    /* Set request for normal mode */
    CAN2SetOperationMode(CAN_IDLE_STOP &
```

```
            CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_NOR &
            CAN_CAPTURE_DIS);

    while (C1CTRLbits.OPMODE != CAN_NORMAL_MODE);
    while (C2CTRLbits.OPMODE != CAN_NORMAL_MODE);

    LED1 = LED_ON;

    /* Load message ID and data into transmit buffer and set transmit
     * request bit */
    CAN1SendMessage(CAN_TX_SID(MSG_SID) &
        /* Send a normal (data) frame and not a remote transmission request */
        CAN_TX_EID_DIS &
        CAN_SUB_NOR_TX_REQ,
        /* EID */
        0,
        data, datalen,
        /* Send with transmit buffer 0 */
        CAN_TXB0
    );

    LED2 = LED_ON;

    /* Wait until transmit buffer 0 has send the message */
    while (!CAN1IsTXReady(CAN_TXB0));

    LED3 = LED_ON;

    /* Wait until receive buffer zero contains a valid message */
    while (!CAN2IsRXReady(CAN_RXB0));

    /* Read received data from receive buffer 0 and store it into user
     * defined dataarray */
    CAN2ReceiveMessage(datareceived, datalen, CAN_RXB0);

    if (datareceived[0] == data[0]) {
        LED4 = LED_ON;
    }

    while (1);

    return 0;
}
```

B.4. Simple AND-coupling module test

B.4.1. couplerModule.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity couplerModule is
    port
    (
        rx_0:    in std_logic;
        rx_1:    in std_logic;
        rx_2:    in std_logic;
        rehRx_0: in std_logic;
        rehRx_1: in std_logic;

        tx_0:    out std_logic;
        tx_1:    out std_logic;
        tx_2:    out std_logic;
        hubTx_0: out std_logic;
        hubTx_1: out std_logic
    );
end couplerModule;


architecture Behavioral of couplerModule is

begin
    tx_0 <= rx_0 and rx_1 and rx_2 and rehRx_0 and rehRx_1;
    tx_1 <= rx_0 and rx_1 and rx_2 and rehRx_0 and rehRx_1;
    tx_2 <= rx_0 and rx_1 and rx_2 and rehRx_0 and rehRx_1;
    hubTx_0 <= rx_0 and rx_1 and rx_2 and rehRx_0 and rehRx_1;
    hubTx_1 <= rx_0 and rx_1 and rx_2 and rehRx_0 and rehRx_1;

end Behavioral;
```

B.4.2. couplerModule.ucf

```
NET "rx_0"  LOC = "H15"  ;
NET "rx_1"  LOC = "H14"  ;
NET "rx_2"  LOC = "G12"  ;
NET "rehRx_0"  LOC = "G16"  ;            # Bank 5
NET "rehRx_1"  LOC = "H13"  ;

NET "tx_0"  LOC = "G15"  ;
NET "tx_1"  LOC = "G14"  ;
NET "tx_2"  LOC = "F14"  ;
NET "hubTx_0"  LOC = "F15"  ;
NET "hubTx_1"  LOC = "G13"  ;
```

B.4.3. msg.h

```
/* Defines the contents of the message transmitted from the transmitting
 * node to the receiving node. */

#ifndef _MSG_H_
#define _MSG_H_


#define MSG_SID 0x0AAA // 010 1010 1010
```

```
/* The DLC field of a CAN frame restricts the values to 0, 1, ... 8 */
#define NUM_PAYLOAD_BYTES 1

#define PAYLOAD_BYTE 0x0A // 0000 1010

#endif /* _MSG_H_ */
```

B.4.4. receiver.c

```
/*
 * Receiving node. Accepts one CAN frame through the CAN1 module.
 */


#include <p30F6014A.h>
#include <can.h>
#define __dsPIC30F6014A__    /* Required for can.h */

#include "../../dspicdem.h"
#include "../../can_aux.h"
#include "../../device_config.h"
#include "../../portd.h"

#include "../msg.h"


void init_CAN1(void)
{
    int i;

    CAN1SetOperationMode(CAN_IDLE_STOP & CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_CONFIG & CAN_CAPTURE_DIS);

    while (C1CTRLbits.OPMODE != CAN_CONFIG_MODE);

    CAN1Initialize(CAN_SYNC_JUMP_WIDTH1 & CAN_BAUD_PRE_SCALE(BRP_PLUS1),
            CAN_WAKEUP_BY_FILTER_DIS &
            CAN_PHASE_SEG1_TQ(CAN_SEG1_TQ) &
            CAN_PHASE_SEG2_TQ(CAN_SEG2_TQ) &
            CAN_PROPAGATIONTIME_SEG_TQ(CAN_PROP_TQ) &
            CAN_SEG2_FREE_PROG & CAN_SAMPLE1TIME);

    for (i = CAN_RXB0; i <= CAN_RXB1; i++) {
        CAN1SetRXMode(i, CAN_RXFUL_CLEAR & CAN_BUF0_DBLBUFFER_DIS);
    }

    /* Setup both of RXB0's filters */
    for (i = 0; i < 2; i++) {
        CAN1SetFilter(i, CAN_FILTER_SID(0) & CAN_RX_EID_DIS, 0);
    }

    /* Mask all bits, ie any message is accepted */
```

```
    CAN1SetMask(0, CAN_MASK_SID(0) & CAN_MATCH_FILTER_TYPE,
            CAN_MASK_EID(0));

    /* We do not configure masks and filters of RXB1 because we don't use
     * RXB1 */
}

int main (void)
{
    int i;
    unsigned char datareceived[NUM_PAYLOAD_BYTES];

    init_portd();
    init_CAN1();

    CAN1SetOperationMode(CAN_IDLE_STOP & CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_NOR & CAN_CAPTURE_DIS);

    while (C1CTRLbits.OPMODE != CAN_NORMAL_MODE);

    LED1 = LED_ON;

    while (!CAN1IsRXReady(CAN_RXB0));

    LED2 = LED_ON;

    CAN1ReceiveMessage(datareceived, NUM_PAYLOAD_BYTES, CAN_RXB0);

    LED3 = LED_ON;

    /* Dummy instruction, needed to turn on LED4. (For some reason if I try
     * to turn on two LEDs, one immediately after the other, then the first
     * one doesn't get turned on) */
    i = 0;

    LED4 = LED_ON;

    for (i = 0; i < NUM_PAYLOAD_BYTES; i++) {
        if (datareceived[i] != PAYLOAD_BYTE) {
            LED4 = LED_OFF;
        }
    }

    while (1);

    return 0;
}
```

B.4.5. transmitter.c

```
/*
 * Transmitting node. Sends one CAN frame through the CAN1 module.
 */
```

```
#include <p30F6014A.h>
#include <can.h>
#define __dsPIC30F6014A__   /* Required for can.h */

#include "../../dspicdem.h"
#include "../../can_aux.h"
#include "../../device_config.h"
#include "../../portd.h"

#include "../msg.h"


void init_CAN1(void)
{
    int i;

    CAN1SetOperationMode(CAN_IDLE_STOP & CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_CONFIG & CAN_CAPTURE_DIS);

    while (C1CTRLbits.OPMODE != CAN_CONFIG_MODE);

    CAN1Initialize(CAN_SYNC_JUMP_WIDTH1 & CAN_BAUD_PRE_SCALE(BRP_PLUS1),
            CAN_WAKEUP_BY_FILTER_DIS &
            CAN_PHASE_SEG1_TQ(CAN_SEG1_TQ) &
            CAN_PHASE_SEG2_TQ(CAN_SEG2_TQ) &
            CAN_PROPAGATIONTIME_SEG_TQ(CAN_PROP_TQ) &
            CAN_SEG2_FREE_PROG & CAN_SAMPLE1TIME);


    /* Configure transmit buffers */
    for (i = CAN_TXB0; i <= CAN_TXB2; i++) {
        CAN1SetTXMode(i, CAN_TX_STOP_REQ & CAN_TX_PRIORITY_HIGH);
    }
}

int main (void)
{
    int i;
    unsigned char data[NUM_PAYLOAD_BYTES];

    init_portd();
    init_CAN1();

    CAN1SetOperationMode(CAN_IDLE_STOP & CAN_MASTERCLOCK_1 &
            CAN_REQ_OPERMODE_NOR & CAN_CAPTURE_DIS);

    while (C1CTRLbits.OPMODE != CAN_NORMAL_MODE);

    LED1 = LED_ON;

    for (i = 0; i < NUM_PAYLOAD_BYTES; i++) {
            data[i] = PAYLOAD_BYTE;
    }
```

```
    CAN1SendMessage(CAN_TX_SID(MSG_SID) & CAN_TX_EID_DIS &
            CAN_SUB_NOR_TX_REQ, 0, data, NUM_PAYLOAD_BYTES,
            CAN_TXB0);

    LED2 = LED_ON;

    /* Wait until transmit buffer 0 has send the message */
    while (!CAN1IsTXReady(CAN_TXB0));

    LED3 = LED_ON;

    /* Dummy instruction, needed to turn on LED4. (For some reason if I try
     * to turn on two LEDs, one immediately after the other, then the first
     * one doesn't get turned on) */
    i = 0;

    LED4 = LED_ON;

    while (1);

    return 0;
}
```

C. Driver source code

C.1. assert.c

```
/*
 * assert.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "led.h"
#include <p30f6014A.h>
#include "interrupts.h"

#ifndef NDEBUG



#define END_OF_STRING 0
#define FILE_NAME_LENGTH_MAX 100

char asserted_file_name[FILE_NAME_LENGTH_MAX + 1] = { END_OF_STRING };
int asserted_line_number = 0;




static void copy_string(
    char *source,
    char *destination
)
{
    int i = 0;

    while (source[i] != END_OF_STRING && i < FILE_NAME_LENGTH_MAX) {
        destination[i] = source[i];
        i++;
    }
    destination[i] = END_OF_STRING;
}



static void wait(void)
{
    int i, j;
```

```
    for (i = 0; i < 10000; i++) {
        for (j = 0; j < 100; j++);
    }
}


void aFailed2(
    char *file_name,
    int line,
    char led_value
)
{
    /* Disable interrupts */
    SET_CPU_IPL(INTERRUPT_PRIORITY_MAX);

    /* Copy the file name and line number where the assert failed to a
     * fixed memory location so that these memory locations can be looked
     * up in a debugger to determine where the assert failed. */
    copy_string(file_name, asserted_file_name);
    asserted_line_number = line;
    init_leds();
    while (1) {
        /* Flash LEDs to show that an assert failed */
        led_display(0x0);
        wait();
        led_display(led_value);
        wait();
    }
}


void aFailed(
    char *file_name,
    int line
)
{
    aFailed2(file_name, line, 0xFF);
}


#endif // NDEBUG
```

C.2. assert.h

```
/*
 * assert.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */
```

```
#ifndef _ASSERT_H_
#define _ASSERT_H_


#ifdef NDEBUG

#define ASSERT(expr) ((void)0)

#else

void aFailed(
    char *file_name,
    int line
);

#define ASSERT(expr) if (expr) {/*Do nothing*/} else\
    aFailed(__FILE__, __LINE__)

void aFailed2(
    char *file_name,
    int line,
    char led_value
);

#define ASSERT2(expr, led_value) if (expr) {/*Do nothing*/} else\
    aFailed2(__FILE__, __LINE__, led_value)

#endif /* NDEBUG */


#endif /* _ASSERT_H_ */
```

C.3. can_controller.c

```
/*
 * can_controller.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include <p30f6014A.h>
#include "can_controller.h"
#include "common.h"
#include "assert.h"
#include "can_frame.h"



/*
 * Controller 1 receive buffer initialization
 */
static struct rx_buffer_struct ctrl1_rx_buffer0 = {
```

```
    .data = &C1RX0B1,
    .SIDbits = &C1RX0SIDbits,
    .DLCbits = &C1RX0DLCbits,
    .type = RX_BUFFER0,
    .CONbitsRX0 = &C1RX0CONbits,
    .CONbitsRX1 = NULL,
    .MaskSIDbits = &C1RXM0SIDbits,
};

static struct rx_buffer_struct ctrl1_rx_buffer1 = {
    .data = &C1RX1B1,
    .SIDbits = &C1RX1SIDbits,
    .DLCbits = &C1RX1DLCbits,
    .type = RX_BUFFER1,
    .CONbitsRX0 = NULL,
    .CONbitsRX1 = &C1RX1CONbits,
    .MaskSIDbits = &C1RXM1SIDbits,
};



/*
 * Controller 1 transmit buffer initialization
 */
static struct tx_buffer_struct ctrl1_tx_buffer0 = {
    .data = &C1TX0B1,
    .SIDbits = &C1TX0SIDbits,
    .DLCbits = &C1TX0DLCbits,
    .CONbits = &C1TX0CONbits
};

struct tx_buffer_struct ctrl1_tx_buffer1 = {
    .data = &C1TX1B1,
    .SIDbits = &C1TX1SIDbits,
    .DLCbits = &C1TX1DLCbits,
    .CONbits = &C1TX1CONbits
};

static struct tx_buffer_struct ctrl1_tx_buffer2 = {
    .data = &C1TX2B1,
    .SIDbits = &C1TX2SIDbits,
    .DLCbits = &C1TX2DLCbits,
    .CONbits = &C1TX2CONbits
};



/*
 * Controller 1 initialization
 */
struct can_controller ctrl1 = {
    .notified_rx = false,
    .notified_tx = false,
    .is_active = true,
```

```
    .INTFbits = &C1INTFbits,
    .CTRLbits = &C1CTRLbits,
    .CFG1bits = &C1CFG1bits,
    .CFG2bits = &C1CFG2bits,
    .INTEbits = &C1INTEbits,

    .rx_buffer[0] = &ctrl1_rx_buffer0,
    .rx_buffer[1] = &ctrl1_rx_buffer1,

    .FilterSIDbits[0] = &C1RXF0SIDbits,
    .FilterSIDbits[1] = &C1RXF1SIDbits,
    .FilterSIDbits[2] = &C1RXF2SIDbits,
    .FilterSIDbits[3] = &C1RXF3SIDbits,
    .FilterSIDbits[4] = &C1RXF4SIDbits,
    .FilterSIDbits[5] = &C1RXF5SIDbits,

    .tx_buffer[0] = &ctrl1_tx_buffer0,
    .tx_buffer[1] = &ctrl1_tx_buffer1,
    .tx_buffer[2] = &ctrl1_tx_buffer2,

    .rx_buffer_loaded = NULL,
};



/*
 * Controller 2 receive buffer initialization
 */
static struct rx_buffer_struct ctrl2_rx_buffer0 = {
    .data = &C2RX0B1,
    .SIDbits = &C2RX0SIDbits,
    .DLCbits = &C2RX0DLCbits,
    .type = RX_BUFFER0,
    .CONbitsRX0 = &C2RX0CONbits,
    .CONbitsRX1 = NULL,
    .MaskSIDbits = &C2RXM0SIDbits,
};

static struct rx_buffer_struct ctrl2_rx_buffer1 = {
    .data = &C2RX1B1,
    .SIDbits = &C2RX1SIDbits,
    .DLCbits = &C2RX1DLCbits,
    .type = RX_BUFFER1,
    .CONbitsRX0 = NULL,
    .CONbitsRX1 = &C2RX1CONbits,
    .MaskSIDbits = &C2RXM1SIDbits,
};



/*
 * Controller 2 transmit buffer initialization
 */
static struct tx_buffer_struct ctrl2_tx_buffer0 = {
```

```
    .data = &C2TX0B1,
    .SIDbits = &C2TX0SIDbits,
    .DLCbits = &C2TX0DLCbits,
    .CONbits = &C2TX0CONbits
};

static struct tx_buffer_struct ctrl2_tx_buffer1 = {
    .data = &C2TX1B1,
    .SIDbits = &C2TX1SIDbits,
    .DLCbits = &C2TX1DLCbits,
    .CONbits = &C2TX1CONbits
};

static struct tx_buffer_struct ctrl2_tx_buffer2 = {
    .data = &C2TX2B1,
    .SIDbits = &C2TX2SIDbits,
    .DLCbits = &C2TX2DLCbits,
    .CONbits = &C2TX2CONbits
};



/*
 * Controller 2 initialization
 */
struct can_controller ctrl2 = {
    .notified_rx = false,
    .notified_tx = false,
    .is_active = true,
    .INTFbits = &C2INTFbits,
    .CTRLbits = &C2CTRLbits,
    .CFG1bits = &C2CFG1bits,
    .CFG2bits = &C2CFG2bits,
    .INTEbits = &C2INTEbits,

    .rx_buffer[0] = &ctrl2_rx_buffer0,
    .rx_buffer[1] = &ctrl2_rx_buffer1,

    .FilterSIDbits[0] = &C2RXF0SIDbits,
    .FilterSIDbits[1] = &C2RXF1SIDbits,
    .FilterSIDbits[2] = &C2RXF2SIDbits,
    .FilterSIDbits[3] = &C2RXF3SIDbits,
    .FilterSIDbits[4] = &C2RXF4SIDbits,
    .FilterSIDbits[5] = &C2RXF5SIDbits,

    .tx_buffer[0] = &ctrl2_tx_buffer0,
    .tx_buffer[1] = &ctrl2_tx_buffer1,
    .tx_buffer[2] = &ctrl2_tx_buffer2,

    .rx_buffer_loaded = NULL,
};


/* The current transmission controller */
```

```
static volatile struct can_controller *tx_controller = &ctrl1;

/* CAN Module Operation Modes, used with the REQOP field of the CAN
 * control and status register (CTRLbits) of ctrl1 and ctrl2. */
typedef enum t_can_mode_enum {
    CAN_MODE_NORMAL = 00,
    CAN_MODE_DISABLE = 01,
    CAN_MODE_LOOPBACK = 02,
    CAN_MODE_LISTEN_ONLY = 03,
    CAN_MODE_CONFIG = 04,
    /* 05 and 06 are reserved in REQOP */
    CAN_MODE_LISTEN_ALL_MSGS = 07
} t_can_mode;

typedef enum {
    CAN_TX_PRIORIY_HIGHEST = 03,
    CAN_TX_PRIORIY_HIGH_INTERMEDIATE = 02,
    CAN_TX_PRIORIY_LOW_INTERMEDIATE = 01,
    CAN_TX_PRIORIY_LOWEST = 00
} t_can_tx_priority;


/* Compile with the appropriate bit rate. For instance, if -DBITRATE921KBPS is
 * passed as an option to the compiler, then the code will be compiled to use a
 * bit rate of 921.25 Kbps for CAN.
 *
 * BRP_VALUE: CAN Baud Rate Prescaler. Valid values are 0, 1, ... 63
 * CAN_PROP_TQ: Length in time quanta for the propagation segment, valid values
 * are 1, 2, ... 8
 * CAN_SEG1_TQ: Length in time quanta for the phase segment 1, valid values are
 * 1, 2, ... 8
 * CAN_SEG2_TQ: Length in time quanta for the phase segment 2, valid values are
 * 1, 2, ... 8
 *
 * The nominal bit rate, NBR, is:
 *   NBR = 1 / NBT
 * where NBT is the nominal bit time. The NBT in turn is:
 *   NBT = NOMINAL_BIT_TIME_TQ * TQ
 * where NOMINAL_BIT_TIME_TQ is the number of time quanta the nominal bit time
 * is made of (defined below) and where TQ is the length of a time quantum. TQ
 * is:
 *   TQ = 2 * (BRP_VALUE + 1) / FCAN
 * Therefore the NBR is:
 *   NBR = NOMINAL_BIT_TIME_TQ * ( 2 * (BRP_VALUE + 1) / FCAN)
 *
 */
#ifdef BITRATE921KBPS
    #define BRP_VALUE 1
    #define CAN_PROP_TQ 2
    #define CAN_SEG1_TQ 3
    #define CAN_SEG2_TQ 2
#elif defined(BITRATE737KBPS)
    #define BRP_VALUE 1
    #define CAN_PROP_TQ 3
```

```
    #define CAN_SEG1_TQ 4
    #define CAN_SEG2_TQ 2
#elif defined(BITRATE670KBPS)
    #define BRP_VALUE 1
    #define CAN_PROP_TQ 4
    #define CAN_SEG1_TQ 4
    #define CAN_SEG2_TQ 2
#elif defined(BITRATE567KBPS)
    #define BRP_VALUE 1
    #define CAN_PROP_TQ 5
    #define CAN_SEG1_TQ 5
    #define CAN_SEG2_TQ 2
#elif defined(BITRATE460KBPS)
    #define BRP_VALUE 1
    #define CAN_PROP_TQ 8
    #define CAN_SEG1_TQ 5
    #define CAN_SEG2_TQ 2
#elif defined(BITRATE230KBPS)
    #define BRP_VALUE 3
    #define CAN_PROP_TQ 8
    #define CAN_SEG1_TQ 5
    #define CAN_SEG2_TQ 2
#else
    #error "Bit rate not specified!"
#endif

#define SYNCHRONOUS_JUMP_WIDTH_TQ 1
/* Sync segment is always one time quantum */
#define CAN_SYNC_TQ 1
/* Number of time quanta the nominal bit time is made of */
#define NOMINAL_BIT_TIME_TQ (CAN_SYNC_TQ + CAN_PROP_TQ + CAN_SEG1_TQ + \
    CAN_SEG2_TQ)




/* Converts a given number of time quanta to a corresponding value which
 * can be assigned to a CAN baud rate configuration register field. For
 * instance, to configure a SJW of 1 TQ the value 0 must be assigned to
 * the SJW field of a CiCFG1 register. */
static unsigned int TQ_to_config_value(
     unsigned int time_quanta
)
{
    return time_quanta - 1;
}



static void enable_can_interrupts(
    struct can_controller *ctrl
)
{
```

```
    /* The invalid message received interrupt and the bus wake up
     * activity interrupt are left disabled. */
    ctrl->INTEbits->IVRIE = 0;
    ctrl->INTEbits->WAKIE = 0;

    /* Enable error interrupt */
    ctrl->INTEbits->ERRIE = 1;

    /* Enable transmit buffer 2 interrupt */
    ctrl->INTEbits->TX2IE = 1;
    /* Enable transmit buffer 1 interrupt */
    ctrl->INTEbits->TX1IE = 1;
    /* Enable transmit buffer 0 interrupt */
    ctrl->INTEbits->TX0IE = 1;

    /* Enable receive buffer 1 interrupt */
    ctrl->INTEbits->RX1IE = 1;
    /* Enable receive buffer 0 interrupt */
    ctrl->INTEbits->RX0IE = 1;
}



static void disable_can_interrupts(
    volatile struct can_controller *const ctrl
)
{
    /* The invalid message received interrupt and the bus wake up
     * activity interrupt are left disabled. */
    ctrl->INTEbits->IVRIE = 0;
    ctrl->INTEbits->WAKIE = 0;

    /* Disable error interrupt */
    ctrl->INTEbits->ERRIE = 0;

    /* Disable transmit buffer 2 interrupt */
    ctrl->INTEbits->TX2IE = 0;
    /* Disable transmit buffer 1 interrupt */
    ctrl->INTEbits->TX1IE = 0;
    /* Disable transmit buffer 0 interrupt */
    ctrl->INTEbits->TX0IE = 0;

    /* Disable receive buffer 1 interrupt */
    ctrl->INTEbits->RX1IE = 0;
    /* Disable receive buffer 0 interrupt */
    ctrl->INTEbits->RX0IE = 0;
}



static void init_acceptance_filters(
    struct can_controller *ctrl
)
{
```

```
    int filter_idx;

    for (filter_idx = 0; filter_idx < ACCEPTANCE_FILTER_COUNT;
        filter_idx++) {
        /* Enable filter for standard identifier and not extended
         * identifier */
        ctrl->FilterSIDbits[filter_idx]->EXIDE = 0;

        /* SID to match doesn't matter because the masks in
         * init_acceptance_filter_masks() have been configured to
         * accept every message. We can therefore initialize the
         * SID field with an arbitrary value (we use 0). */
        ctrl->FilterSIDbits[filter_idx]->SID = 0;
    }
}



/*
 * The message acceptance filters and masks determine if a message in the
 * message assembly buffer (MAB) should be loaded into one of the receive
 * buffers. The filters and masks are applied to the message identifier.
 * The mask determines which bits of the identifier should be examined and
 * the filters contain values to which those bits are compared. The bits
 * from the identifier that are masked, i.e. the corresponding mask bit is
 * zero, will always be accepted by the filters. The bits that are not
 * masked, i.e. the corresponding mask bit is one, will be accepted if
 * there is a match with the corresponding filter bit. If all the bits are
 * accepted then the message is accepted and loaded from the MAB into one
 * of the receive buffers.
 *
 * Messages whose identifier match filters RXF0 or RXF1 are loaded into
 * receive buffer 0 (RXB0), messages whose identifier match any of the
 * filters RXF2 through RXF5 are loaded into receive buffer 1 (RXB1). The
 * mask RXM0 is used with filters RXF0 and RXF1 and the mask RXM1 is used
 * with filters RXF2–RXF5.
 */
static void init_acceptance_filter_masks(
    struct can_controller *ctrl
)
{
    int rx_buffer_idx;

    for (rx_buffer_idx = 0; rx_buffer_idx < RX_BUFFER_COUNT;
        rx_buffer_idx++) {
        /* Configure acceptance filter mask to accept all messages,
         * i.e. all the masks' bits are set to zero therefore all
         * messages are accepted independently of the value of the
         * acceptance filters. */
        ctrl->rx_buffer[rx_buffer_idx]->MaskSIDbits->SID = 0;

        /* Match only message types (SID or EID) as determined by
         * the EXIDE bit in the corresponding filters */
        ctrl->rx_buffer[rx_buffer_idx]->MaskSIDbits->MIDE = 1;
```

```
    }
}


static void init_controller(
    struct can_controller *ctrl
)
{
    int tx_buffer_idx = 0;

    /* Stop CAN module when device enters idle mode. */
    ctrl->CTRLbits->CSIDL = 1;
    /* FCAN clock is FCY (instruction cycle clock) instead of
     * FOSC = 4 x FCY */
    ctrl->CTRLbits->CANCKS = 1;
    /* Don't generate a capture signal whenever a valid frame has been
     * accepted */
    ctrl->CTRLbits->CANCAP = 0;

    /* Set configuration mode */
    ctrl->CTRLbits->REQOP = CAN_MODE_CONFIG;

    /* Wait until the CAN module has entered configuration mode */
    while (ctrl->CTRLbits->OPMODE != CAN_MODE_CONFIG);

    ctrl->CFG1bits->SJW =
        TQ_to_config_value(SYNCHRONOUS_JUMP_WIDTH_TQ);

    ctrl->CFG1bits->BRP = BRP_VALUE;

    /* CAN bus line filter is not used for wake-up */
    ctrl->CFG2bits->WAKFIL = 0;
    /* The length of Phase Segment 2 is Freely programmable */
    ctrl->CFG2bits->SEG2PHTS = 1;
    /* Bus line is sampled once at the sample point */
    ctrl->CFG2bits->SAM = 0;

    /* Set number of time quanta to use for propagation segment */
    ctrl->CFG2bits->PRSEG = TQ_to_config_value(CAN_PROP_TQ);
    /* Set number of time quanta to use for segment 1 */
    ctrl->CFG2bits->SEG1PH = TQ_to_config_value(CAN_SEG1_TQ);
    /* Set number of time quanta to use for segment 2 */
    ctrl->CFG2bits->SEG2PH = TQ_to_config_value(CAN_SEG2_TQ);

    /*
     * Configure transmit buffers
     */
    for (tx_buffer_idx = 0; tx_buffer_idx < TX_BUFFER_COUNT;
        tx_buffer_idx++) {
        /* Clear transmit request bit */
        ctrl->tx_buffer[tx_buffer_idx]->CONbits->TXREQ = 0;
        /* TODO: Should all transmit buffers have the same
         * priority? */
```

```
            ctrl->tx_buffer[tx_buffer_idx]->CONbits->TXPRI =
                CAN_TX_PRIORIY_HIGHEST;
        }

        /*
         * Configure receive buffers
         */
        /* Clear receive full status bit */
        ctrl->rx_buffer[0]->CONbitsRX0->RXFUL = 0;
        ctrl->rx_buffer[1]->CONbitsRX1->RXFUL = 0;
        /* Disable double buffer, i.e. no receive buffer 0 overflow to
         * receive buffer 1 */
        ctrl->rx_buffer[0]->CONbitsRX0->DBEN = 0;

        /* TODO: Allow the user to override the receive and transmit
         * buffer configuration by adding corresponding functions to
         * recancentrate.h */

        init_acceptance_filter_masks(ctrl);

        /* TODO: Allow the user to override the mask configuration and to
         * also configure the acceptance filter. */

        init_acceptance_filters(ctrl);

        enable_can_interrupts(ctrl);

        /* Set normal mode */
        ctrl->CTRLbits->REQOP = CAN_MODE_NORMAL;

    /* The simulator does not model the CAN module */
    #ifndef SIMULATOR
        /* Wait until the CAN module has entered normal mode */
        while (ctrl->CTRLbits->OPMODE != CAN_MODE_NORMAL);
    #endif
    }



    void init_can_controllers(void)
    {
        /*
         * Restrictions on the CAN bit time segments (see the dsPIC30F Family
         * Reference Manual for details):
         */
        ASSERT(CAN_PROP_TQ + CAN_SEG1_TQ >= CAN_SEG2_TQ);
        ASSERT(CAN_SEG2_TQ > SYNCHRONOUS_JUMP_WIDTH_TQ);
        ASSERT(8 <= NOMINAL_BIT_TIME_TQ);
        ASSERT(NOMINAL_BIT_TIME_TQ <= 25);
        ASSERT(0 <= BRP_VALUE);
        ASSERT(BRP_VALUE <= 63);

        init_controller(&ctrl1);
        init_controller(&ctrl2);
```

```
}



bool is_transmission_controller(
    const volatile struct can_controller *const ctrl
)
{
    return (tx_controller == ctrl);
}



bool is_active(
    const volatile struct can_controller *const ctrl
)
{
    return ctrl->is_active;
}



void shutdown(
    volatile struct can_controller *const ctrl
)
{
    disable_can_interrupts(ctrl);
    ctrl->CTRLbits->ABAT = 1;
}



void mark_as_inactive(
    volatile struct can_controller *const ctrl
)
{
    ctrl->is_active = false;
}



static bool has_tx_pending(
    struct tx_buffer_struct *tx_buffer
)
{
    return tx_buffer->CONbits->TXREQ;
}



/* Returns a transmit buffer of ctrl which has no transmission pending.
 * If no free transmit buffer is available it returns NULL. */
static struct tx_buffer_struct* get_free_tx_buffer(
    volatile struct can_controller *const ctrl
```

```
)
{
    struct tx_buffer_struct *current_tx_buffer;
    struct tx_buffer_struct *free_tx_buffer = NULL;
    /* Transmission buffer index */
    unsigned int tx_buffer_idx;
    bool free_tx_buffer_found = false;

    /* Search for a free transmit buffer, i.e. a transmit buffer which has
     * no transmission pending */
    tx_buffer_idx = 0;
    while (!free_tx_buffer_found && tx_buffer_idx < TX_BUFFER_COUNT) {
        current_tx_buffer = ctrl->tx_buffer[tx_buffer_idx];
        if (!has_tx_pending(current_tx_buffer)) {
            free_tx_buffer = current_tx_buffer;
            free_tx_buffer_found = true;
        }
        tx_buffer_idx++;
    }

    if (free_tx_buffer_found) {
        return free_tx_buffer;
    } else {
        return NULL;
    }
}



/* Instructs one of ctrl's free transmission buffers to transmit the frame
 * frame_to_tx */
void request_tx(
    volatile struct can_controller *const ctrl,
    struct can_frame *frame_to_tx
)
{
    struct tx_buffer_struct *tx_buffer_to_use;

    ASSERT(is_active(ctrl));

    tx_buffer_to_use = get_free_tx_buffer(ctrl);

    if (tx_buffer_to_use == NULL) {
        /* TODO: What shall we do if there is no free tx buffer?
         * For the moment we ASSERT(false) */
        ASSERT(false);
    }

    tx_buffer_to_use->SIDbits->SID5_0 = frame_to_tx->identifier & 0x003F;
    tx_buffer_to_use->SIDbits->SID10_6 = frame_to_tx->identifier & 0x07C0;

    /* Copy frame_to_tx's data to the transmit buffer's data register */
    copy_data(frame_to_tx->data, frame_to_tx->length,
        (unsigned char*) tx_buffer_to_use->data);
```

```
    tx_buffer_to_use->DLCbits->DLC = frame_to_tx->length;

    /* Signal transmit buffer to enqueue the loaded frame for transmission.
     * The transmission will start when the transmit buffer detects that
     * the medium is available. */
    tx_buffer_to_use->CONbits->TXREQ = 1;
}



void release_rx_buffer(
    volatile struct can_controller *const ctrl,
    volatile struct rx_buffer_struct *buf_to_release
)
{
    if (buf_to_release->type == RX_BUFFER0) {
        ASSERT(ctrl->rx_buffer[0] == buf_to_release);
        buf_to_release->CONbitsRX0->RXFUL = 0;
    } else if (buf_to_release->type == RX_BUFFER1) {
        ASSERT(ctrl->rx_buffer[1] == buf_to_release);
        buf_to_release->CONbitsRX1->RXFUL = 0;
    } else {
        /* Invalid receive buffer type */
        ASSERT(false);
    }
    set_rx_event_causing_rx_buffer(ctrl, NULL);
}



/* Returns the receive buffer of controller 'ctrl' where the last received
 * frame has been stored and which, thus, caused the last CAN combined
 * interrupt for 'ctrl' that was triggered due to a reception. */
volatile struct rx_buffer_struct* get_rx_event_causing_rx_buffer(
    volatile struct can_controller *const ctrl
)
{
    ASSERT(is_active(ctrl));
    ASSERT(ctrl->rx_buffer_loaded != NULL);

    return ctrl->rx_buffer_loaded;
}



void set_rx_event_causing_rx_buffer(
    volatile struct can_controller *const ctrl,
    volatile struct rx_buffer_struct *const buf
)
{
    ctrl->rx_buffer_loaded = buf;
}
```

```
void read_frame(
    volatile struct rx_buffer_struct *const buffer_to_read,
    volatile struct can_frame *const output_frame
)
{
    int num_bytes_to_read = buffer_to_read->DLCbits->DLC;

    /* Assert the frame has a standard identifier and not an
     * extended identifier */
    ASSERT(buffer_to_read->SIDbits->RXIDE == 0);

    output_frame->identifier = buffer_to_read->SIDbits->SID;

    /* Copy contents of the receive buffer to the data field of
     * output_frame */
    copy_data((unsigned char*) buffer_to_read->data, num_bytes_to_read,
        output_frame->data);

    output_frame->length = num_bytes_to_read;
}



bool tx_buffer0_irq_occurred(
    const volatile struct can_controller *const ctrl
)
{
    ASSERT(is_active(ctrl));

    return ctrl->INTFbits->TX0IF;
}



void clear_tx_buffer0_irq(
    const volatile struct can_controller *const ctrl
)
{
    ctrl->INTFbits->TX0IF = 0;
}



bool tx_buffer1_irq_occurred(
    const volatile struct can_controller *const ctrl
)
{
    ASSERT(is_active(ctrl));

    return ctrl->INTFbits->TX1IF;
}
```

```
void clear_tx_buffer1_irq(
    const volatile struct can_controller *const ctrl
)
{
    ctrl->INTFbits->TX1IF = 0;
}



bool tx_buffer2_irq_occurred(
    const volatile struct can_controller *const ctrl
)
{
    ASSERT(is_active(ctrl));

    return ctrl->INTFbits->TX2IF;
}



void clear_tx_buffer2_irq(
    const volatile struct can_controller *const ctrl
)
{
    ctrl->INTFbits->TX2IF = 0;
}



bool rx_buffer0_irq_occured(
    const volatile struct can_controller *const ctrl
)
{
    return ctrl->INTFbits->RX0IF;
}



void clear_rx_buffer0_irq(
    const volatile struct can_controller *const ctrl
)
{
    ctrl->INTFbits->RX0IF = 0;
}



bool rx_buffer1_irq_occured(
    const volatile struct can_controller *const ctrl
)
{
    return ctrl->INTFbits->RX1IF;
```

```
}


void clear_rx_buffer1_irq(
    const volatile struct can_controller *const ctrl
)
{
    ctrl->INTFbits->RX1IF = 0;
}


bool error_irq_occurred(
    const volatile struct can_controller *const ctrl
)
{
    ASSERT(is_active(ctrl));

    return (ctrl->INTFbits->ERRIF);
}


bool notified_rx(
    const volatile struct can_controller *const ctrl
)
{
    return (ctrl->notified_rx == true);
}


void set_notified_rx(
    volatile struct can_controller *const ctrl,
    const bool b
)
{
    ASSERT(is_active(ctrl));

    ctrl->notified_rx = b;
}


bool notified_tx(
    const volatile struct can_controller *const ctrl
)
{
    return (ctrl->notified_tx == true);
}
```

```
void set_notified_tx(
    volatile struct can_controller *const ctrl,
    const bool b
)
{
    ASSERT(is_active(ctrl));

    ctrl->notified_tx = b;
}



void set_transmission_controller(
    volatile struct can_controller *const ctrl
)
{
    tx_controller = ctrl;
}



volatile struct can_controller *get_transmission_controller(void)
{
    return tx_controller;
}
```

C.4. can_controller.h

```
/*
 * can_controller.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _CAN_CONTROLLER_H_
#define _CAN_CONTROLLER_H_



#include <p30f6014A.h>
#include "can_frame.h"
#include "common.h"


/* The receive buffer 0 and the receive buffer 1 of a dsPIC's CAN controller
 * have different control registers, therefore we distinguish between the two
 * types of receive buffers. */
typedef enum {
    RX_BUFFER0,
    RX_BUFFER1
} t_rx_buffer_type;
```

```
/* CAN receive buffer */
struct rx_buffer_struct {
    /* First word of the receive buffer. The second, third and fourth word
     * are in contiguous memory locations following the first word */
    volatile unsigned int *data;
    /* Pointer to receive buffer standard identifier register */
    volatile CxRXxSIDBITS *SIDbits;
    /* Pointer to receive buffer data length code register */
    volatile CxRXxDLCBITS *DLCbits;
    t_rx_buffer_type type;
    /* Pointer to receive buffer control register */
    /* ... for receive buffers of type RX_BUFFER0 */
    volatile CxRX0CONBITS *CONbitsRX0;
    /* ... for receive buffers of type RX_BUFFER1 */
    volatile CxRX1CONBITS *CONbitsRX1;
    /* Pointer to the SID acceptance filter mask register */
    volatile CxRXMxSIDBITS *MaskSIDbits;
    /* TODO: Add masks for extended ids? */
};



/* CAN transmit buffer */
struct tx_buffer_struct {
    /* First word of the transmit buffer. The second, third and fourth word
     * are in contiguous memory locations following the first word */
    volatile unsigned int *data;
    /* Pointer to CAN Standard Identifier register */
    volatile CxTXxSIDBITS *SIDbits;
    /* Pointer to transmit buffer data length code register */
    volatile CxTXxDLCBITS *DLCbits;
    /* Pointer to transmit buffer control register */
    volatile CxTXxCONBITS *CONbits;
};



/* Number of receive buffers per CAN controller.
 * The dsPIC's CAN controllers have two visible receive buffers (the third
 * buffer is the message assembly buffer (MAB) and is not directly
 * accessible). */
#define RX_BUFFER_COUNT 2
/* Number of transmit buffers per CAN controller. */
#define TX_BUFFER_COUNT 3
/* Number of acceptance filters per CAN controller */
#define ACCEPTANCE_FILTER_COUNT 6



/*
 * Keeps information about a CAN controller
 */
struct can_controller {
    /* True if the CAN controller notified the reception of a CAN frame */
```

```
    bool notified_rx;
    /* True if the CAN controller notified the transmission of a CAN frame
     */
    bool notified_tx;
    /* False when the CAN controller is isolated, true when it is in use */
    bool is_active;
    /* Pointer to CAN interrupt flag status register */
    volatile CxINTFBITS *INTFbits;
    /* Pointer to CAN control and status register */
    volatile CxCTRLBITS *CTRLbits;
    /* Pointer to CAN baud rate configuration register 1 */
    volatile CxCFG1BITS *CFG1bits;
    /* Pointer to CAN baud rate configuration register 2 */
    volatile CxCFG2BITS *CFG2bits;
    /* Pointer to CAN interrupt enable register */
    volatile CxINTEBITS *INTEbits;
    /* The CAN controller's receive buffers */
    struct rx_buffer_struct *rx_buffer[RX_BUFFER_COUNT];
    /* Pointers to the SID acceptance filter registers */
    volatile CxRXFxSIDBITS *FilterSIDbits[ACCEPTANCE_FILTER_COUNT];
    /* The CAN controller's transmit buffers */
    struct tx_buffer_struct *tx_buffer[TX_BUFFER_COUNT];
    /* The rx_buffer that contains the received frame (NULL if no buffer
     * has a frame) */
    volatile struct rx_buffer_struct *rx_buffer_loaded;
};



/*
 * Operations of the CAN controller ADT
 */

/* Initializes both CAN controllers */
void init_can_controllers(void);

/* Returns true if 'ctrl' is the controller that has the transmission
 * controller role assigned to; returns false otherwise */
bool is_transmission_controller(
    const volatile struct can_controller *const ctrl
);

/* Returns true if 'ctrl' has not been quarantined by the qua routine */
bool is_active(
    const volatile struct can_controller *const ctrl
);

/* Shuts 'ctrl' down, disabling all its interrupts and aborting all its
 * transmissions */
void shutdown(
    volatile struct can_controller *const ctrl
);

/* Marks 'ctrl' as having been quarantined */
```

```
void mark_as_inactive(
    volatile struct can_controller *const ctrl
);

/* Instructs one of ctrl's free transmission buffers to transmit the frame
 * 'frame_to_tx' */
void request_tx(
    volatile struct can_controller *const ctrl,
    struct can_frame *frame_to_tx
);

/* Returns true if an error occured at 'ctrl' */
bool error_irq_occurred(
    const volatile struct can_controller *const ctrl
);

/* Returns true if set_notified_rx(ctrl, true) has previously been called */
bool notified_rx(
    const volatile struct can_controller *const ctrl
);

/* Used by the CAN event tracker to indicate that 'ctrl' notified a reception
 */
void set_notified_rx(
    volatile struct can_controller *const ctrl,
    const bool b
);

/* Returns true if set_notified_tx(ctrl, true) has previously been called */
bool notified_tx(
    const volatile struct can_controller *const ctrl
);

/* Used by the CAN event tracker to indicate that 'ctrl' notified a
 * transmission */
void set_notified_tx(
    volatile struct can_controller *const ctrl,
    const bool b);

/* Assigns the transmission controller role to 'ctrl' */
void set_transmission_controller(
    volatile struct can_controller *const ctrl
);

/* Returns a pointer to the controller that is currently the transmission
 * controller */
volatile struct can_controller *get_transmission_controller(void);




/*
 * Operations of the nested reception buffer ADT
```

```
 */

/* Releases the reception buffer 'buf_to_release' of 'ctrl' so that it is free
 * to receive a new frame */
void release_rx_buffer(
    volatile struct can_controller *const ctrl,
    volatile struct rx_buffer_struct *buf_to_release
);

/* Returns the receive buffer of controller 'ctrl' where the last received
 * frame has been stored */
volatile struct rx_buffer_struct* get_rx_event_causing_rx_buffer(
    volatile struct can_controller *const ctrl
);

/* Marks the receive buffer 'buf' as being the one which received the last
 * frame */
void set_rx_event_causing_rx_buffer(
    volatile struct can_controller *const ctrl,
    volatile struct rx_buffer_struct *const buf
);

/* Reads the frame containd within 'buffer_to_read' into 'frame_read' */
void read_frame(
    volatile struct rx_buffer_struct *const buffer_to_read,
    volatile struct can_frame *const frame_read
);

/* Returns true if reception buffer 0 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool rx_buffer0_irq_occured(
    const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that reception buffer 0 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_rx_buffer0_irq(
    const volatile struct can_controller *const ctrl
);

/* Returns true if reception buffer 1 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool rx_buffer1_irq_occured(
    const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that reception buffer 1 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_rx_buffer1_irq(
    const volatile struct can_controller *const ctrl
);
```

```
/*
 * Operations of the nested transmission buffer ADT
 */

/* Returns true if transmission buffer 0 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool tx_buffer0_irq_occurred(
    const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that transmission buffer 0 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_tx_buffer0_irq(
    const volatile struct can_controller *const ctrl
);

/* Returns true if transmission buffer 1 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool tx_buffer1_irq_occurred(
    const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that transmission buffer 1 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_tx_buffer1_irq(
    const volatile struct can_controller *const ctrl
);

/* Returns true if transmission buffer 2 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool tx_buffer2_irq_occurred(
    const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that transmission buffer 2 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_tx_buffer2_irq(
    const volatile struct can_controller *const ctrl
);




#endif /* _CAN_CONTROLLER_H_ */
```

C.5. can_frame.c

```
/*
 * can_frame.c
 *
```

```
 * Written by David Gessner <davidges@gmail.com>
 */

#include "can_frame.h"
#include "assert.h"




bool equals_frame(
    const volatile struct can_frame *const frame1,
    const volatile struct can_frame *const frame2
)
{
    int i;

    if (frame1->identifier != frame2->identifier) {
        return false;
    }

    if (frame1->length != frame2->length) {
        return false;
    }

    for (i = 0; i < frame1->length; i++) {
        if (frame1->data[i] != frame2->data[i]) {
            return false;
        }
    }
    return true;
}



void copy_frame(
    const volatile struct can_frame *const src,
    volatile struct can_frame *const dst
)
{
    *dst = *src;
}



void copy_data(
    volatile unsigned const char *const input_buffer,
    int num_bytes_to_copy,
    volatile unsigned char *const output_buffer
)
{
    int byte_idx = 0;

    ASSERT(0 <= num_bytes_to_copy &&
```

```
        num_bytes_to_copy <= CAN_PAYLOAD_LEN_MAX);

    for (byte_idx = 0; byte_idx < num_bytes_to_copy; byte_idx++) {
        output_buffer[byte_idx] = input_buffer[byte_idx];
    }
}
```

C.6. can_frame.h

```
/*
 * can_frame.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _CAN_FRAME_H_
#define _CAN_FRAME_H_



#include "common.h"



/* According to the CAN specification a CAN data frame can carry at most 8
 * bytes */
#define CAN_PAYLOAD_LEN_MAX 8




struct can_frame {
    unsigned char data[CAN_PAYLOAD_LEN_MAX];
    unsigned char length;
    unsigned int identifier;
};

bool equals_frame(
    const volatile struct can_frame *const frame1,
    const volatile struct can_frame *const frame2
);

void copy_frame(
    const volatile struct can_frame *const src,
    volatile struct can_frame *const dst
);



void copy_data(
    volatile unsigned const char *const input_buffer,
    int num_bytes_to_copy,
```

```
    volatile unsigned char *const output_buffer
);


#endif /* _CAN_FRAME_H_ */
```

C.7. common.h

```
/*
 * common.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _COMMON_H_
#define _COMMON_H_

typedef enum bool_enum {
    false = 0,
    true
} bool;

#define NULL 0

#endif /* _COMMON_H_ */
```

C.8. device_config.h

```
/*
 * device_config.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef __DEVICE_CONFIG_H_
#define __DEVICE_CONFIG_H_



#include <p30f6014A.h>



/******** DEVICE CONFIGURATION ********/

/* Oscillator configuration
 *
 * Oscillators provided by the dsPICDEM board:
 *     - Y1: 7.37 MHz
 *     - Y2: 32.768 KHz
 *     - Y3: socket for external oscillator
 * dsPIC30F6014A internal oscillators:
```

```
 *      - FRC: 7.37 MHz
 *      - LPRC: 512 KHz
 */
_FOSC(
    /* Clock switching and fail safe clock monitor off, i.e. do not detect
     * clock failures and do not switch over to internal FRC oscillator. */
    CSW_FSCM_OFF &
    /* Use crystall oscillator (i.e. Y1 on dsPICDEM board) multiplying
     * the clock speed by 16 with a Phase Locked-Loop. The result is that
     * the oscillator has a frequency of 16 * 7.37 = 117.92 MHz, i.e. FOSC =
     * 117.92 MHz. But FOSC is not the frequency used for the instruction
     * cycle. The instruction cycle's frequency is FCY = FOSC/4 = 29.48
     * MHz. */
    XT_PLL16);

/* Watchdog timer configuration = watchdog timer off. */
_FWDT(WDT_OFF);

/* Reset configuration */
_FBORPOR(
    /* Enable brown out at 20 volts. */
    PBOR_ON & BORV_20 &
    /* Power up timer = 64ms, gives the oscillator time to start and
     * stabilize. */
    PWRT_64 &
    /* Master clear reset enabled, i.e. use the MCLR pin as a reset signal
     * instead of using it as an IO pin. Pulling the MCLR pin low will
     * reset the dsPIC and start execution from 0x000. */
    MCLR_EN);

/* General Code Segment configuration = Disable Code Protection */
_FGS(CODE_PROT_OFF);


#endif /* ___DEVICE_CONFIG_H_ */
```

C.9. interrupts.c

```
/*
 * interrupts.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include <p30f6014A.h>
#include "interrupts.h"
#include "common.h"
#include "assert.h"
```

```
void set_interrupt_priority(
    int priority,
    t_interrupt interrupt
)
{
    ASSERT(INTERRUPT_PRIORITY_MIN <= priority);
    ASSERT(priority <= INTERRUPT_PRIORITY_MAX);

    switch(interrupt) {
    case HW_INTERRUPT_CAN1:
        IPC6bits.C1IP = priority;
        break;
    case HW_INTERRUPT_CAN2:
        IPC9bits.C2IP = priority;
        break;
    case HW_INTERRUPT_TIMER1:
        IPC0bits.T1IP = priority;
        break;
    case SW_INTERRUPT_CAN1_TX_EVENT:
        SW_INTERRUPT_CAN1_TX_EVENT_PRIORITY = priority;
        break;
    case SW_INTERRUPT_CAN1_RX_EVENT:
        SW_INTERRUPT_CAN1_RX_EVENT_PRIORITY = priority;
        break;
    case SW_INTERRUPT_CAN1_ERROR_WARNING:
        SW_INTERRUPT_CAN1_ERROR_WARNING_PRIORITY = priority;
        break;
    case SW_INTERRUPT_CAN2_TX_EVENT:
        SW_INTERRUPT_CAN2_TX_EVENT_PRIORITY = priority;
        break;
    case SW_INTERRUPT_CAN2_RX_EVENT:
        SW_INTERRUPT_CAN2_RX_EVENT_PRIORITY = priority;
        break;
    case SW_INTERRUPT_CAN2_ERROR_WARNING:
        SW_INTERRUPT_CAN2_ERROR_WARNING_PRIORITY = priority;
        break;
    default:
        /* We should never get here */
        ASSERT(false);
        break;
    }
}


void enable_interrupt(
    t_interrupt interrupt
)
{
    switch(interrupt) {
    case HW_INTERRUPT_CAN1:
        IEC1bits.C1IE = 1;
        break;
    case HW_INTERRUPT_CAN2:
```

```
            IEC2bits.C2IE = 1;
            break;
        case HW_INTERRUPT_TIMER1:
            IEC0bits.T1IE = 1;
            break;
        case SW_INTERRUPT_CAN1_TX_EVENT:
            SW_INTERRUPT_CAN1_TX_EVENT_IE = 1;
            break;
        case SW_INTERRUPT_CAN1_RX_EVENT:
            SW_INTERRUPT_CAN1_RX_EVENT_IE = 1;
            break;
        case SW_INTERRUPT_CAN1_ERROR_WARNING:
            SW_INTERRUPT_CAN1_ERROR_WARNING_IE = 1;
            break;
        case SW_INTERRUPT_CAN2_TX_EVENT:
            SW_INTERRUPT_CAN2_TX_EVENT_IE = 1;
            break;
        case SW_INTERRUPT_CAN2_RX_EVENT:
            SW_INTERRUPT_CAN2_RX_EVENT_IE = 1;
            break;
        case SW_INTERRUPT_CAN2_ERROR_WARNING:
            SW_INTERRUPT_CAN2_ERROR_WARNING_IE = 1;
            break;
        default:
            /* We should never get here */
            ASSERT(false);
            break;
        }
}


void disable_interrupt(
    t_interrupt interrupt
)
{
    switch(interrupt) {
    case HW_INTERRUPT_CAN1:
        IEC1bits.C1IE = 0;
        break;
    case HW_INTERRUPT_CAN2:
        IEC2bits.C2IE = 0;
        break;
    case HW_INTERRUPT_TIMER1:
        IEC0bits.T1IE = 0;
        break;
    case SW_INTERRUPT_CAN1_TX_EVENT:
        SW_INTERRUPT_CAN1_TX_EVENT_IE = 0;
        break;
    case SW_INTERRUPT_CAN1_RX_EVENT:
        SW_INTERRUPT_CAN1_RX_EVENT_IE = 0;
        break;
    case SW_INTERRUPT_CAN1_ERROR_WARNING:
        SW_INTERRUPT_CAN1_ERROR_WARNING_IE = 0;
```

```
            break;
        case SW_INTERRUPT_CAN2_TX_EVENT:
            SW_INTERRUPT_CAN2_TX_EVENT_IE = 0;
            break;
        case SW_INTERRUPT_CAN2_RX_EVENT:
            SW_INTERRUPT_CAN2_RX_EVENT_IE = 0;
            break;
        case SW_INTERRUPT_CAN2_ERROR_WARNING:
            SW_INTERRUPT_CAN2_ERROR_WARNING_IE = 0;
            break;
        default:
            /* We should never get here */
            ASSERT(false);
            break;
    }
}



void set_interrupt_flag(
    t_interrupt interrupt
)
{
    switch(interrupt) {
    case HW_INTERRUPT_CAN1:
    case HW_INTERRUPT_CAN2:
    case HW_INTERRUPT_TIMER1:
        /* Should only be set by hardware, not through software */
        ASSERT(false);
        break;
    case SW_INTERRUPT_CAN1_TX_EVENT:
        SW_INTERRUPT_CAN1_TX_EVENT_IF = 1;
        break;
    case SW_INTERRUPT_CAN1_RX_EVENT:
        SW_INTERRUPT_CAN1_RX_EVENT_IF = 1;
        break;
    case SW_INTERRUPT_CAN1_ERROR_WARNING:
        SW_INTERRUPT_CAN1_ERROR_WARNING_IF = 1;
        break;
    case SW_INTERRUPT_CAN2_TX_EVENT:
        SW_INTERRUPT_CAN2_TX_EVENT_IF = 1;
        break;
    case SW_INTERRUPT_CAN2_RX_EVENT:
        SW_INTERRUPT_CAN2_RX_EVENT_IF = 1;
        break;
    case SW_INTERRUPT_CAN2_ERROR_WARNING:
        SW_INTERRUPT_CAN2_ERROR_WARNING_IF = 1;
        break;
    default:
        /* We should never get here */
        ASSERT(false);
        break;
    }
}
```

```
void clear_interrupt_flag(
    t_interrupt interrupt
)
{
    switch(interrupt) {
    case HW_INTERRUPT_CAN1:
        IFS1bits.C1IF = 0;
        break;
    case HW_INTERRUPT_CAN2:
        IFS2bits.C2IF = 0;
        break;
    case HW_INTERRUPT_TIMER1:
        IFS0bits.T1IF = 0;
        break;
    case SW_INTERRUPT_CAN1_TX_EVENT:
        SW_INTERRUPT_CAN1_TX_EVENT_IF = 0;
        break;
    case SW_INTERRUPT_CAN1_RX_EVENT:
        SW_INTERRUPT_CAN1_RX_EVENT_IF = 0;
        break;
    case SW_INTERRUPT_CAN1_ERROR_WARNING:
        SW_INTERRUPT_CAN1_ERROR_WARNING_IF = 0;
        break;
    case SW_INTERRUPT_CAN2_TX_EVENT:
        SW_INTERRUPT_CAN2_TX_EVENT_IF = 0;
        break;
    case SW_INTERRUPT_CAN2_RX_EVENT:
        SW_INTERRUPT_CAN2_RX_EVENT_IF = 0;
        break;
    case SW_INTERRUPT_CAN2_ERROR_WARNING:
        SW_INTERRUPT_CAN2_ERROR_WARNING_IF = 0;
        break;
    default:
        /* We should never get here */
        ASSERT(false);
        break;
    }
}
```

C.10. interrupts.h

```
/*
 * interrupts.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _INTERRUPTS_H_
#define _INTERRUPTS_H_
```

```
#define INTERRUPT_PRIORITY_MIN 0
#define INTERRUPT_PRIORITY_MAX 7


typedef enum t_interrupt_enum {
    /* Interrupts generated by hardware */
    HW_INTERRUPT_CAN1,
    HW_INTERRUPT_CAN2,
    HW_INTERRUPT_TIMER1,

    /* Interrupts generated through software by the
     * CAN event tracker */
    SW_INTERRUPT_CAN1_TX_EVENT,
    SW_INTERRUPT_CAN1_RX_EVENT,
    SW_INTERRUPT_CAN1_ERROR_WARNING,
    SW_INTERRUPT_CAN2_TX_EVENT,
    SW_INTERRUPT_CAN2_RX_EVENT,
    SW_INTERRUPT_CAN2_ERROR_WARNING
} t_interrupt;

/* The dsPIC microcontroller does not provide software interrupts. As a
 * workaround we use interrupts belonging to unused hardware peripherals as
 * software interrupts */

/* The following interrupts are ordered from highest natural order priority to
 * lowest natural order priority, i.e. they are ordered by the priority they
 * will have if all 6 interrupts have the same user selectable priority. */

/* Use UART1 Receiver Interrupt as CAN transmit event on CAN controller 1 */
#define _CAN1TxEventInterrupt _U1RXInterrupt
#define SW_INTERRUPT_CAN1_TX_EVENT_PRIORITY IPC2bits.U1RXIP
#define SW_INTERRUPT_CAN1_TX_EVENT_IE IEC0bits.U1RXIE
#define SW_INTERRUPT_CAN1_TX_EVENT_IF IFS0bits.U1RXIF

/* Use UART1 Transmitter Interrupt as CAN transmit event on CAN controller 2 */
#define _CAN2TxEventInterrupt _U1TXInterrupt
#define SW_INTERRUPT_CAN2_TX_EVENT_PRIORITY IPC2bits.U1TXIP
#define SW_INTERRUPT_CAN2_TX_EVENT_IE IEC0bits.U1TXIE
#define SW_INTERRUPT_CAN2_TX_EVENT_IF IFS0bits.U1TXIF

/* Use External Interrupt 1 as CAN receive event on CAN controller 1 */
#define _CAN1RxEventInterrupt _INT1Interrupt
#define SW_INTERRUPT_CAN1_RX_EVENT_PRIORITY IPC4bits.INT1IP
#define SW_INTERRUPT_CAN1_RX_EVENT_IE IEC1bits.INT1IE
#define SW_INTERRUPT_CAN1_RX_EVENT_IF IFS1bits.INT1IF

/* Use External Interrupt 2 as CAN receive event on CAN controller 2 */
#define _CAN2RxEventInterrupt _INT2Interrupt
#define SW_INTERRUPT_CAN2_RX_EVENT_PRIORITY IPC5bits.INT2IP
#define SW_INTERRUPT_CAN2_RX_EVENT_IE IEC1bits.INT2IE
```

```
#define SW_INTERRUPT_CAN2_RX_EVENT_IF IFS1bits.INT2IF

/* Use External Interrupt 3 as Error Warning event on CAN controller 1 */
#define _CAN1ErrWarnEventInterrupt _INT3Interrupt
#define SW_INTERRUPT_CAN1_ERROR_WARNING_PRIORITY IPC9bits.INT3IP
#define SW_INTERRUPT_CAN1_ERROR_WARNING_IE IEC2bits.INT3IE
#define SW_INTERRUPT_CAN1_ERROR_WARNING_IF IFS2bits.INT3IF

/* Use External Interrupt 4 as Error Warning event on CAN controller 2 */
#define _CAN2ErrWarnEventInterrupt _INT4Interrupt
#define SW_INTERRUPT_CAN2_ERROR_WARNING_PRIORITY IPC9bits.INT4IP
#define SW_INTERRUPT_CAN2_ERROR_WARNING_IE IEC2bits.INT4IE
#define SW_INTERRUPT_CAN2_ERROR_WARNING_IF IFS2bits.INT4IF



#define enable_interrupt_nesting() asm("BCLR INTCON1, #15")


void set_interrupt_priority(
    int priority,
    t_interrupt interrupt
);

void enable_interrupt(
    t_interrupt interrupt
);

void disable_interrupt(
    t_interrupt interrupt
);

void set_interrupt_flag(
    t_interrupt interrupt
);

void clear_interrupt_flag(
    t_interrupt interrupt
);



#endif /* _INTERRUPTS_H_ */
```

C.11. led.c

```
/*
 * led.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include <p30f6014A.h>
```

```
void init_leds(void)
{
    /* The LEDs are connected to pins 4-7 of port D on the dspic demo
     * board. */

    /* LEDs initially turned off */
    PORTD = 0;
    /* TRISD configures each pin of port D as either an input (1) or an
     * output (0). Set RD7 to RD4, i.e. LED4 to LED1, as outputs */
    TRISD = 0xFF0F;
}



/* Lights the 4 LEDs to display 'b' in binary */
void led_display(char b)
{
    PORTD &= 0xFF0F;
    PORTD |= ((b & 0x000F)<<4);
}
```

C.12. led.h

```
/*
 * led.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _LED_H_
#define _LED_H_



void init_leds(void);

void led_display(char b);

#endif /* _LED_H_ */
```

C.13. quaroutine.c

```
/*
 * quaroutine.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "interrupts.h"
#include "can_controller.h"
#include "assert.h"
```

```
#include "tx_timer.h"

#ifdef PROFILE
#include "profiler.h"
#endif



extern volatile struct can_controller ctrl1, ctrl2;

extern bool tx_pending;
extern struct can_frame frame_to_tx;
extern bool active_controller_available;




static inline void signal_no_controllers_available_to_user_software(void)
{
    active_controller_available = false;
}



static inline void quarantine(
    /* Controller to be quarantined */
    volatile struct can_controller *const this_ctrl,
    /* The other controller */
    volatile struct can_controller *const other_ctrl
)
{
    mark_as_inactive(this_ctrl);
    shutdown(this_ctrl);

    /* TODO: reset controller? */

    if (is_transmission_controller(this_ctrl)) {
        disable_tx_timer();
        if (is_active(other_ctrl)) {
            set_transmission_controller(other_ctrl);
            if (tx_pending && !notified_tx(this_ctrl)) {
                request_tx(other_ctrl, &frame_to_tx);
                reset_to_zero_tx_timer();
                enable_tx_timer();
            }
        } else {
            signal_no_controllers_available_to_user_software();
        }
    } else {
        /* other_ctrl is the transmission controller, therefore it
         * must be active */
        ASSERT(is_active(other_ctrl));
```

```
        /* Do nothing */
    }
}



static inline void handle_error_warning_or_tx_timeout(
    /* Controller which caused the error warning or transmission timeout */
    volatile struct can_controller *const this_ctrl,
    /* The other controller */
    volatile struct can_controller *const other_ctrl
)
{
    /* Only one of the controllers must be the transmission controller
     */
    ASSERT(is_transmission_controller(this_ctrl) ||
        is_transmission_controller(other_ctrl));
    ASSERT(!(is_transmission_controller(this_ctrl) &&
        is_transmission_controller(other_ctrl)));

    if (is_active(this_ctrl)) {
        quarantine(this_ctrl, other_ctrl);
    } else {
        /* this_ctrl has already been quarantined */
        /* Do nothing */
    }
}



/*
 * This ISR is invoked through a software interrupt set by
 * track_can_event_and_invoke_isr() when it detects that an error warning
 * was signaled by the CAN1 controller.
 */
void __attribute__((__interrupt__, no_auto_psv))
_CAN1ErrWarnEventInterrupt(void)
{
#ifdef PROFILE
    profiler_start_timer5();
#endif
    handle_error_warning_or_tx_timeout(&ctrl1, &ctrl2);

    clear_interrupt_flag(SW_INTERRUPT_CAN1_ERROR_WARNING);
#ifdef PROFILE
    profiler_stop_timer5();
#endif
}



/*
 * This ISR is invoked through a software interrupt set by
 * track_can_event_and_invoke_isr() when it detects that an error warning
```

```
 * was signaled by the CAN2 controller.
 */
void __attribute__((__interrupt__, no_auto_psv))
_CAN2ErrWarnEventInterrupt(void)
{
#ifdef PROFILE
    profiler_start_timer5();
#endif
    handle_error_warning_or_tx_timeout(&ctrl2, &ctrl1);

    clear_interrupt_flag(SW_INTERRUPT_CAN2_ERROR_WARNING);
#ifdef PROFILE
    profiler_stop_timer5();
#endif
}



/*
 * This ISR is invoked when the transmission timeout has been reached.
 */
void __attribute__((__interrupt__, no_auto_psv)) _T1Interrupt(void)
{
    if (is_transmission_controller(&ctrl1)) {
        handle_error_warning_or_tx_timeout(&ctrl1, &ctrl2);
    } else {
        handle_error_warning_or_tx_timeout(&ctrl2, &ctrl1);
    }
}
```

C.14. rxroutine.c

```
/*
 * rxroutine.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "interrupts.h"
#include "common.h"
#include "assert.h"
#include "can_controller.h"

#ifdef PROFILE
#include "profiler.h"
#endif



extern volatile struct can_controller ctrl1, ctrl2;


/*
```

```
 * Variables used by the ReCANcentrate API and the media management ISRs to
 * communicate to each other:
 */

extern struct can_frame frame_to_tx;
extern struct can_frame frame_to_deliver;
extern bool tx_pending;
extern bool tx_success;
extern bool rx_data_available;


/*
 * Variables which are shared between media management ISRs to coordinate their
 * actions:
 */

/* When a delivery event occurs and both of the node's CAN controllers are
 * active, then two interrupts occur, one for each controller. The first
 * invocation of the corresponding media management ISR sets this boolean
 * variable to true so that the second ISR invocation knows that the delivery
 * event has already been handled by the first ISR invocation. */
bool delivery_event_handled = false;

/* Counter that indicates the number of times a notification has been omitted
 * from either the transmitting or the non-transmitting controller for a
 * frame that is to be transmitted. */
unsigned int omission_count = 0;

/* The maximum permitted value for omission_count. If omission_count
 * reaches OMISSION_COUNT_MAX we consider the omissions not due to a CAN
 * inconsistency scenario (i.e. a scenario where some controllers, of any node,
 * accept a frame while others reject it) but due to a crash of a controller. */
const unsigned int OMISSION_COUNT_MAX = 3;




/* Let the program that uses this driver know that a transmission was
 * successful */
void signal_tx_success_to_user_software(void)
{
    tx_success = true;
    tx_pending = false;
}



void ack_own_successful_tx(void)
{
    /* Coordinate with the ISR which is invoked due to an interrupt
     * from the other controller, i.e. let it know that the delivery
     * event has already been handled. */
    delivery_event_handled = true;
```

```
    omission_count = 0;

    signal_tx_success_to_user_software();
}


static inline void signal_rx_success_to_user_software(void)
{
    rx_data_available = true;
}


static inline void handle_reception_from_other_node(
    const struct can_frame *const received_frame,
    volatile struct can_controller *const other_ctrl
)
{
    ASSERT(!delivery_event_handled);
    if (notified_rx(other_ctrl)) {
        /* Make sure that when handle_receive_event() is executed
         * again (due to a receive event notification from other_ctrl)
         * it knows that the received frame has already been managed */
        delivery_event_handled = true;
    } else {
        /* Do nothing. */
        /* other_ctrl has not notified reception, so we don't set
         * delivery_event_handled to true because no ISR will be
         * executed due to other_ctrl which could reset it to false */
    }
    copy_frame(received_frame, &frame_to_deliver);
    signal_rx_success_to_user_software();
}


static inline void handle_tx_notification_omission(
    /* The controller that omitted the transmission notification */
    volatile struct can_controller *const other_ctrl
)
{
    /* In the comments below this_ctrl refers to the controller which
     * notified of the reception of a frame and through whose ISR we got
     * here. */

    /* Precondition: other_ctrl did not notify a transmission. */
    ASSERT(!notified_tx(other_ctrl));

    /* Possible reasons for other_ctrl omiting the notification of a
     * transmission:
     * 1) Both this_ctrl and other_ctrl detected an error in the last bit
     * of the EOF. The receiving controller, i.e. this_ctrl, accepted the
```

```
 * frame because in CAN receiving controllers must accept frames with
 * the last bit erroneous. On the other hand the transmitting
 * controller, i.e. other_ctrl, did not consider the transmission as
 * successful and therefore omitted the notification.
 * 2) other_ctrl detected an error after the CRC but could not
 * globalize the error because a fault in its uplink prevented it from
 * sending an error frame. Therefore other_ctrl considered the
 * transmission as failed while all other controllers (this_ctrl and
 * the ones from the other nodes) did not receive
 * the error frame and therefore considered the received frame as
 * correct.
 * 3) A fault occured in this_ctrl's downlink after the ACK bit,
 * preventing it from receiving an error frame and leading it to
 * erroneously accept the frame. On the other hand other_ctrl detected
 * the error frame and correctly considered the transmission as failed.
 * 4) other_ctrl crashed in such a way that it was able to transmit the
 * frame but it was not able to notify of the transmission, e.g. it
 * crashed after sending the CRC.
 *
 * All 4 cases can cause inconsistencies, i.e. some controllers receive
 * the frame while others do not. As OMISSION_COUNT_MAX must have a
 * value larger than or equal to the maximum possible number of
 * consecutive inconsistencies, if the followinng ASSERT fails then the
 * value of OMISSION_COUNT_MAX is probably to small. */
ASSERT(omission_count < OMISSION_COUNT_MAX);

/* To solve the inconsistencies the frame has to be retransmitted,
 * possibly leading some controllers to receive a duplicate frame but,
 * and this is the important point, making sure that every controller
 * gets at least one copy.
 *
 * If other_ctrl omitted for reason 1), other_ctrl will retransmit the
 * frame.
 *
 * If other_ctrl omitted for reason 2) and the fault on other_ctrl's
 * uplink was transient then other_ctrl will retransmit and the
 * retransmission will probably be successful (if not, it will be
 * handled again by the ISRs). If other_ctrl omitted for reason 2) and
 * the fault on other_ctrl's uplink is permanent then other_ctrl will
 * be quarantined (either because it reaches the error warning limit or
 * because the transmit timeout expires) and this_ctrl will become the
 * new transmission controller and it will be instructed to retransmit
 * the frame.
 *
 * If reason 3) caused the omission then other_ctrl will retransmit the
 * frame. If the downlink fault in reason 3) is permanent then any
 * further retransmissions will not be notified by this_ctrl leading
 * omission_count to reach OMISSION_COUNT_MAX, point at which the
 * transmission routine will consider the transmission as successful.
 * Further transmissions through other_ctrl will also be considered
 * successful as omission_count will not be reset.
 *
 * If reason 4) caused the omission then the transmission timeout will
 * expire leading other_ctrl to be quarantined and this_ctrl to become
```

```
     * the new transmission controller. Furthermore the quarantine routine
     * will instruct the new transmission controller to retransmit the
     * frame.
     *
     * Whatever caused the omission of other_ctrl, the frame will be
     * retransmitted without any further actions here. All we have to do
     * therefore is to increase omission_count. */
    omission_count++;
}



static inline void handle_frame_reception(
    volatile struct can_controller *const other_ctrl,
    volatile struct rx_buffer_struct *const rx_buffer_to_read
)
{
    struct can_frame received_frame;
    bool received_our_own_frame;

    read_frame(rx_buffer_to_read, &received_frame);

    /* If other_ctrl is active it should have notified of a reception or
     * transmission by now (unless there was a fault). */

    if (notified_tx(other_ctrl)) {
        ack_own_successful_tx();
    } else {
        /* Check why other_ctrl did not notify a transmission */
        if (!tx_pending) {
            /* Everything OK, we don't have anything to transmit
             * therefore the frame was sent by another node and
             * consequently other_ctrl was not supposed to notify a
             * transmission */
            handle_reception_from_other_node(&received_frame,
                other_ctrl);
        } else {
            received_our_own_frame = equals_frame(&received_frame,
                &frame_to_tx);
            if (!received_our_own_frame) {
                /* Everything OK, although a transmission is
                 * pending, the frame received was not sent by
                 * other_ctrl */
                handle_reception_from_other_node(
                    &received_frame, other_ctrl);
            } else {
                /* Fault detected, other_ctrl should have
                 * notified a transmission */
                handle_tx_notification_omission(other_ctrl);
            }
        }
    }
}
```

```
/*
 * Handles a receive event notified by this_ctrl.
 */
static inline void handle_receive_event(
    volatile struct can_controller *const this_ctrl,
    volatile struct can_controller *const other_ctrl
)
{
    volatile struct rx_buffer_struct *rx_buffer_loaded;

    ASSERT(notified_rx(this_ctrl));

    /* Reset this_ctrl's receive notified flag for the next delivery event
     * (it has been previously set by track_can_event_and_invoke_isr()). */
    set_notified_rx(this_ctrl, false);

    /* Determine in which receive buffer the received frame has been stored
     */
    rx_buffer_loaded = get_rx_event_causing_rx_buffer(this_ctrl);

    /* If a media management ISR already handled the current delivery event
     * */
    if (delivery_event_handled) {
        /* Reset for the next delivery event */
        delivery_event_handled = false;
    } else {
        handle_frame_reception(other_ctrl, rx_buffer_loaded);
    }

    release_rx_buffer(this_ctrl, rx_buffer_loaded);
}



/*
 * This ISR is invoked through a software interrupt set by
 * track_can_event_and_invoke_isr() when it detects that a frame was received
 * on the CAN1 controller.
 */
void __attribute__((__interrupt__, no_auto_psv)) _CAN1RxEventInterrupt(void)
{
#ifdef PROFILE
    profiler_start_timer3();
#endif
    /* ACK software interrupt set by track_can_event_and_invoke_isr() */
    clear_interrupt_flag(SW_INTERRUPT_CAN1_RX_EVENT);

    handle_receive_event(&ctrl1, &ctrl2);
#ifdef PROFILE
    profiler_stop_timer3();
#endif
}
```

```
/*
 * This ISR is invoked through a software interrupt set by
 * track_can_event_and_invoke_isr() when it detects that a frame was received
 * on the CAN2 controller.
 */
void __attribute__((__interrupt__, no_auto_psv)) _CAN2RxEventInterrupt(void)
{
#ifdef PROFILE
    profiler_start_timer3();
#endif
    /* ACK software interrupt set by track_can_event_and_invoke_isr() */
    clear_interrupt_flag(SW_INTERRUPT_CAN2_RX_EVENT);

    handle_receive_event(&ctrl2, &ctrl1);
#ifdef PROFILE
    profiler_stop_timer3();
#endif
}
```

C.15. rxroutine.h

```
/*
 * rxroutine.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

void ack_own_successful_tx(void);

void signal_tx_success_to_user_software(void);
```

C.16. tracker.c

```
/*
 * tracker.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

/*
 * CAN event tracker
 *
 * Each non-faulty CAN controller generates an interrupt for every delivery
 * event (i.e. a reception or a transmission of a frame).
 */

#include "interrupts.h"
#include "common.h"
#include "can_controller.h"
#include "assert.h"
```

```
#ifdef PROFILE
#include "profiler.h"
#endif



extern volatile struct can_controller ctrl1;
extern volatile struct can_controller ctrl2;




/*
 * Keeps track of which particular CAN interrupt occurred and invokes through a
 * software interrupt a corresponding media management ISR
 * (_CAN1RxEventInterrupt, _CAN1TxEventInterrupt, _CAN1ErrWarnEventInterrupt,
 * _CAN2RxEventInterrupt, _CAN2TxEventInterrupt or _CAN2ErrWarnEventInterrupt)
 * to handle the particular CAN interrupt.
 *
 * Keeping track of the CAN interrupts is needed so that a media management ISR
 * handling an interrupt from one controller can know whether an interrupt
 * occured on the other controller and of what type it was.
 *
 * The media management ISRs have lower priority than _C1Interrupt and
 * _C2Interrupt so that _C1Interrupt or _C2Interrupt can interrupt the media
 * management ISRs. Also note that because of _C1Interrupt and _C2Interrupt's
 * higher priority the media management ISR invoked will not be nested but
 * executed after _C1Interrupt (or _C2Interrupt respectively) finishes.
 */
static inline void track_can_event_and_invoke_isr(
    volatile struct can_controller *const this_ctrl,
    t_interrupt tx_event_interrupt,
    t_interrupt rx_event_interrupt,
    t_interrupt error_warning_interrupt
)
{
    /* precondition: A CAN combined interrupt occured and we arrived here
     * through _C1Interrupt or _C2Interrupt. */

    /* Discern what exactly caused the CAN interrupt. */

    if (tx_buffer0_irq_occurred(this_ctrl)) {
        /* Assert only one transmission controller notified */
        ASSERT(!tx_buffer1_irq_occurred(this_ctrl) &&
            !tx_buffer2_irq_occurred(this_ctrl));

        /* Keep track of the fact that a transmission occurred on
         * this_ctrl. */
        set_notified_tx(this_ctrl, true);

        /* Generate an interrupt which will invoke (after the current
         * ISR finishes) a media management ISR to handle the transmit
```

```
         * event that occurred on this_ctrl. The invoked ISR will be
         * _CAN1TxEventInterrupt (if this_ctrl is ctrl1) or
         * _CAN2TxEventInterrupt (if this_ctrl is ctrl2). */
        set_interrupt_flag(tx_event_interrupt);

        /* Acknowledge transmit buffer 0 transmitted interrupt */
        clear_tx_buffer0_irq(this_ctrl);
    } else if (tx_buffer1_irq_occurred(this_ctrl)) {
        /* Assert only one transmission controller notified */
        ASSERT(!tx_buffer0_irq_occurred(this_ctrl) &&
            !tx_buffer2_irq_occurred(this_ctrl));

        /* Keep track of the fact that a transmission occurred on
         * this_ctrl. */
        set_notified_tx(this_ctrl, true);

        /* Generate an interrupt which will invoke (after the current
         * ISR finishes) a media management ISR to handle the transmit
         * event that occurred on this_ctrl. The invoked ISR will be
         * _CAN1TxEventInterrupt (if this_ctrl is ctrl1) or
         * _CAN2TxEventInterrupt (if this_ctrl is ctrl2). */
        set_interrupt_flag(tx_event_interrupt);

        /* Acknowledge transmit buffer 0 transmitted interrupt */
        clear_tx_buffer1_irq(this_ctrl);
    } else if (tx_buffer2_irq_occurred(this_ctrl)) {
        /* Assert only one transmission controller notified */
        ASSERT(!tx_buffer0_irq_occurred(this_ctrl) &&
            !tx_buffer1_irq_occurred(this_ctrl));

        /* Keep track of the fact that a transmission occurred on
         * this_ctrl. */
        set_notified_tx(this_ctrl, true);

        /* Generate an interrupt which will invoke (after the current
         * ISR finishes) a media management ISR to handle the transmit
         * event that occurred on this_ctrl. The invoked ISR will be
         * _CAN1TxEventInterrupt (if this_ctrl is ctrl1) or
         * _CAN2TxEventInterrupt (if this_ctrl is ctrl2). */
        set_interrupt_flag(tx_event_interrupt);

        /* Acknowledge transmit buffer 0 transmitted interrupt */
        clear_tx_buffer2_irq(this_ctrl);
    } else if (rx_buffer0_irq_occured(this_ctrl)) {
        /* Assert there is only one receive buffer loaded */
        ASSERT(!rx_buffer1_irq_occured(this_ctrl));

        /* Keep track of the fact that a reception occurred on
         * this_ctrl. */
        set_notified_rx(this_ctrl, true);

        /* Let the rxroutine ISR know which receive buffer to read */
        set_rx_event_causing_rx_buffer(this_ctrl,
            this_ctrl->rx_buffer[0]);
```

```
        /* Generate an interrupt which will invoke (after the current
         * ISR finishes) the rxroutine ISR. The invoked ISR will be
         * _CAN1RxEventInterrupt (if this_ctrl is ctrl1) or
         * _CAN2RxEventInterrupt (if this_ctrl is ctrl2). */
        set_interrupt_flag(rx_event_interrupt);

        /* Acknowledge receive buffer 0 loaded interrupt */
        clear_rx_buffer0_irq(this_ctrl);
    } else if (rx_buffer1_irq_occured(this_ctrl)) {
        /* Assert there is only one receive buffer loaded */
        ASSERT(!rx_buffer0_irq_occured(this_ctrl));

        /* Keep track of the fact that a reception occurred on
         * this_ctrl. */
        set_notified_rx(this_ctrl, true);

        /* Let the rxroutine ISR know which receive buffer to read */
        set_rx_event_causing_rx_buffer(this_ctrl,
            this_ctrl->rx_buffer[1]);

        /* Generate an interrupt which will invoke (after the current
         * ISR finishes) the rxroutine ISR. The invoked ISR will be
         * _CAN1RxEventInterrupt (if this_ctrl is ctrl1) or
         * _CAN2RxEventInterrupt (if this_ctrl is ctrl2). */
        set_interrupt_flag(rx_event_interrupt);

        /* Acknowledge receive buffer 1 loaded interrupt */
        clear_rx_buffer1_irq(this_ctrl);
    }

    /* An error warning can happen just after a reception or transmission,
     * so when the tracker is invoked both a transmission/reception and
     * an error warning may be pending. */

    if (error_irq_occurred(this_ctrl)) {
        /* Keeping track of the fact that an error occurred is not
         * necessary. */

        /* Generate an interrupt which will invoke (after the current
         * ISR finishes) a media management ISR to handle the error
         * warning that occurred on this_ctrl. The invoked ISR will be
         * _CAN1ErrWarnEventInterrupt (if this_ctrl is ctrl1) or
         * _CAN2ErrWarnEventInterrupt (if this_ctrl is ctrl2). */
        set_interrupt_flag(error_warning_interrupt);
    }
}



/*
 * Combined CAN1 ISR
 */
void __attribute__((__interrupt__, no_auto_psv)) _C1Interrupt(void)
```

```
{
#ifdef PROFILE
    profiler_start_timer2();
#endif
    /* Acknowledge the CAN1 combined interrupt */
    clear_interrupt_flag(HW_INTERRUPT_CAN1);

    track_can_event_and_invoke_isr(&ctrl1, SW_INTERRUPT_CAN1_TX_EVENT,
        SW_INTERRUPT_CAN1_RX_EVENT, SW_INTERRUPT_CAN1_ERROR_WARNING);
#ifdef PROFILE
    profiler_stop_timer2();
#endif
}



/*
 * Combined CAN2 ISR
 */
void __attribute__((__interrupt__, no_auto_psv)) _C2Interrupt(void)
{
#ifdef PROFILE
    profiler_start_timer2();
#endif
    /* Acknowledge the CAN2 combined interrupt */
    clear_interrupt_flag(HW_INTERRUPT_CAN2);

    track_can_event_and_invoke_isr(&ctrl2, SW_INTERRUPT_CAN2_TX_EVENT,
        SW_INTERRUPT_CAN2_RX_EVENT, SW_INTERRUPT_CAN2_ERROR_WARNING);
#ifdef PROFILE
    profiler_stop_timer2();
#endif
}
```

C.17. txroutine.c

```
/*
 * txroutine.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "interrupts.h"
#include "can_frame.h"
#include "can_controller.h"
#include "common.h"
#include "assert.h"
#include "tx_timer.h"
#include "rxroutine.h"

#ifdef PROFILE
#include "profiler.h"
#endif
```

```
extern volatile struct can_controller ctrl1, ctrl2;

extern struct can_frame frame_to_tx;

extern bool delivery_event_handled;

extern unsigned int omission_count;

extern const unsigned int OMISSION_COUNT_MAX;




static inline void handle_rx_notification_omission(
    /* The transmission controller */
    volatile struct can_controller *const this_ctrl
)
{
    if (omission_count < OMISSION_COUNT_MAX) {
        omission_count++;

        /* Possible reasons why other_ctrl, i.e. the non-transmission
         * controller, omitted the notification of a reception:
         * 1) Errors at the EOF led the non-transmission controller to
         * reject the frame while the transmission controller
         * considered the transmission correct.
         * 2) The non-transmission controller or a controller of
         * another node sent an error frame, but due to a fault in the
         * transmission controller's downlink the transmission
         * controller did not receive the error frame and it
         * erroneously notified a transmission. On the other hand the
         * non-transmission controller correctly discarded the frame
         * and therefore didn't notify.
         * 3) The non-transmission controller detected an error but
         * wasn't able to globalize it because of a fault in its uplink.
         * Therefore it discarded the frame while all other controllers
         * accepted it.
         * 4) The non-transmission controller crashed but an ACK bit
         * was sent by another node or by the non-transmission
         * controller before it crashed.
         *
         * These scenarios may have led to inconsistencies, i.e. some
         * nodes accepting the frame and others discarding it.
         * Therefore we carry out a retransmission to be sure that all
         * nodes get at least one copy of the frame. */
        request_tx(this_ctrl, &frame_to_tx);
        reset_to_zero_tx_timer();
    } else {
        /* There have been too many omissions. The omissions therefore
         * cannot be due to potential CAN inconsistency scenarios but
         * have to be due to a crash of the non-transmission controller
```

```
         * or a permanent failure at the non-transmission controller's
         * link. Carrying out yet another retransmission doesn't make
         * sense as the non-transmission controller won't be able to
         * receive it anyway and the controllers of the other nodes
         * will most certainly have received the frame already due to
         * the previous retransmissions. We therefore consider the
         * transmission a success. */
        signal_tx_success_to_user_software();

        /* TODO: Mark other_ctrl as inactive? */
    }
}



static inline void handle_frame_transmission(
    volatile struct can_controller *const this_ctrl,
    volatile struct can_controller *const other_ctrl
)
{
    /* If other_ctrl is active it should have notified of a reception by
     * now (unless there was a fault). */

    if (notified_rx(other_ctrl)) {
        ack_own_successful_tx();
    } else {
        handle_rx_notification_omission(this_ctrl);
    }
}



static inline void handle_transmit_event(
    volatile struct can_controller *const this_ctrl,
    volatile struct can_controller *const other_ctrl
)
{
    ASSERT(notified_tx(this_ctrl));

    /* Reset this_ctrl's transmit notified flag for the next delivery event
     * (it has been previously set by track_can_event_and_invoke_isr()) */
    set_notified_tx(this_ctrl, false);

    /* Cancel the transmission timeout */
    disable_tx_timer();

    /* If a media management ISR (i.e. _CAN1RxEventInterrupt or
     * _CAN2RxEventInterrupt) already handled the current delivery event */
    if (delivery_event_handled) {
        /* Reset for the next delivery event */
        delivery_event_handled = false;
    } else {
        handle_frame_transmission(this_ctrl, other_ctrl);
    }
```

```
}



/*
 * This ISR is invoked through a software interrupt set by
 * track_can_event_and_invoke_isr() when it detects that a frame was
 * transmitted on the CAN1 controller.
 */
void __attribute__((__interrupt__, no_auto_psv)) _CAN1TxEventInterrupt(void)
{
#ifdef PROFILE
    profiler_start_timer4();
#endif
    /* ACK software interrupt set by track_can_event_and_invoke_isr() */
    clear_interrupt_flag(SW_INTERRUPT_CAN1_TX_EVENT);

    handle_transmit_event(&ctrl1, &ctrl2);
#ifdef PROFILE
    profiler_stop_timer4();
#endif
}



/*
 * This ISR is invoked through a software interrupt set by
 * track_can_event_and_invoke_isr() when it detects that a frame was
 * transmitted on the CAN2 controller.
 */
void __attribute__((__interrupt__, no_auto_psv)) _CAN2TxEventInterrupt(void)
{
#ifdef PROFILE
    profiler_start_timer4();
#endif
    /* ACK software interrupt set by track_can_event_and_invoke_isr() */
    clear_interrupt_flag(SW_INTERRUPT_CAN2_TX_EVENT);

    handle_transmit_event(&ctrl2, &ctrl1);
#ifdef PROFILE
    profiler_stop_timer4();
#endif
}
```

C.18. tx_timer.c

```
/*
 * tx_timer.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "interrupts.h"
```

```
#include <p30f6014A.h>

/* TODO: choose sensible value */
static unsigned int tx_timeout = 0xFFFF;




static inline void configure_timer1(void)
{
    /* timer1 clock source is internal clock, i.e. FCY = FOSC/4 */
    T1CONbits.TCS = 0;
    /* Do not prescale the timer1 clock source */
    T1CONbits.TCKPS = 0;
    /* Do not use the gated time accumulation mode */
    T1CONbits.TGATE = 0;
    /* In idle mode, stop timer */
    T1CONbits.TSIDL = 1;
    /* Assign timeout to timer1 period register */
    PR1 = tx_timeout;
}



void enable_tx_timer(void)
{
    configure_timer1();
    /* Enable timer1, i.e. the transmission timer */
    T1CONbits.TON = 1;
    enable_interrupt(HW_INTERRUPT_TIMER1);
}



void disable_tx_timer(void)
{
    disable_interrupt(HW_INTERRUPT_TIMER1);
    /* Disable timer1, i.e. the transmission timer */
    T1CONbits.TON = 0;
    clear_interrupt_flag(HW_INTERRUPT_TIMER1);
}



void reset_to_zero_tx_timer(void)
{
    TMR1 = 0;
}
```

C.19. tx_timer.h

```
/*
```

```
 * tx_timer.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */



void enable_tx_timer(void);
void disable_tx_timer(void);
void reset_to_zero_tx_timer(void);
```

D. API source code

D.1. recancentrate.c

```
/*
 * recancentrate.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include <p30f6014A.h>
#include "tx_timer.h"
#include "can_frame.h"
#include "can_controller.h"
#include "interrupts.h"
#include "recancentrate.h"


/*
 * Variables used by the user software and the media management ISRs to
 * communicate to each other:
 */

/* If the node is a transmitter it holds the frame to be transmitted */
struct can_frame frame_to_tx;

/* Message to deliver to the software that uses this driver */
struct can_frame frame_to_deliver;

/* Indicates if the transmission of a frame is pending, i.e. it indicates if
 * the user software requested the transmission of a frame and the frame has
 * not yet been transmited successfully. */
bool tx_pending = false;

/* True if the driver was able to carry out a transmission successfully. */
bool tx_success = false;

/* True if there is data from another node which has been received but which has
 * not been read by the user software. */
bool rx_data_available = false;

/* True if at least one CAN controller is available */
bool active_controller_available = true;
```

```
/* Assign priorities to ReCANcentrate relevant interrupts */
static void assign_interrupt_priorities(void)
{
    /* The priorities assigned to the interrupts must be lower than
     * INTERRUPT_PRIORITY_MAX because we reserve INTERRUPT_PRIORITY_MAX
     * for the CPU to enter an uninterruptable state (although still
     * interruptable by traps) */
    int hw_interrupt_priority = INTERRUPT_PRIORITY_MAX - 1;
    int sw_interrupt_priority = hw_interrupt_priority - 1;

    /* Higher priority for CAN and timer 1 interrupts */
    set_interrupt_priority(hw_interrupt_priority, HW_INTERRUPT_CAN1);
    set_interrupt_priority(hw_interrupt_priority, HW_INTERRUPT_CAN2);
    set_interrupt_priority(hw_interrupt_priority, HW_INTERRUPT_TIMER1);

    /* Lower priority for software interrupts */
    set_interrupt_priority(sw_interrupt_priority,
        SW_INTERRUPT_CAN1_TX_EVENT);
    set_interrupt_priority(sw_interrupt_priority,
        SW_INTERRUPT_CAN1_RX_EVENT);
    set_interrupt_priority(sw_interrupt_priority,
        SW_INTERRUPT_CAN1_ERROR_WARNING);
    set_interrupt_priority(sw_interrupt_priority,
        SW_INTERRUPT_CAN2_TX_EVENT);
    set_interrupt_priority(sw_interrupt_priority,
        SW_INTERRUPT_CAN2_RX_EVENT);
    set_interrupt_priority(sw_interrupt_priority,
        SW_INTERRUPT_CAN2_ERROR_WARNING);
}


/* Enable ReCANcentrate relevant interrupts */
static void enable_interrupts(void)
{
    /* old interrupt priority level */
    int old_ipl;

    /* The to be enabled interrupts have a priority lower than
     * INTERRUPT_PRIORITY_MAX. Therefore setting the CPU's priority level
     * to INTERRUPT_PRIORITY_MAX makes sure that the CPU cannot be
     * interrupted by just enabled interrupts while the other interrupts
     * are still being enabled. This is necessary because all interrupts
     * must be available at the same time to coordinate the different ISRs.
     */
    SET_AND_SAVE_CPU_IPL(old_ipl, INTERRUPT_PRIORITY_MAX);

    enable_interrupt(HW_INTERRUPT_CAN1);
    enable_interrupt(SW_INTERRUPT_CAN1_TX_EVENT);
    enable_interrupt(SW_INTERRUPT_CAN1_RX_EVENT);
    enable_interrupt(SW_INTERRUPT_CAN1_ERROR_WARNING);
```

```
    enable_interrupt(HW_INTERRUPT_CAN2);
    enable_interrupt(SW_INTERRUPT_CAN2_TX_EVENT);
    enable_interrupt(SW_INTERRUPT_CAN2_RX_EVENT);
    enable_interrupt(SW_INTERRUPT_CAN2_ERROR_WARNING);

    RESTORE_CPU_IPL(old_ipl);
}



void init_recancentrate_driver(void)
{
    init_can_controllers();
    enable_interrupt_nesting();

    assign_interrupt_priorities();
    enable_interrupts();
}



static void create_frame(
    unsigned int frame_identifier,
    unsigned char *frame_data,
    unsigned char frame_length
)
{
    frame_to_tx.identifier = frame_identifier;
    copy_data(frame_data, frame_length, frame_to_tx.data);
    frame_to_tx.length = frame_length;
}



void request_recancentrate_tx(
    unsigned int frame_identifier,
    unsigned char *frame_data,
    unsigned char frame_length,
    t_recancentrate_status *output_tx_status
)
{
    /* old interrupt priority level */
    int old_ipl;
    volatile struct can_controller *tx_ctrl;

    if (tx_pending) {
        *output_tx_status = RECANCENTRATE_STATUS_TX_ALREADY_PENDING;
        return;
    }

    create_frame(frame_identifier, frame_data, frame_length);

    /* Raising the CPU's priority makes sure that the value of tx_pending
     * and tx_success and the fact of which controller is the transmission
```

```
     * controller won't be changed by an ISR while the following code is
     * executed. */
    SET_AND_SAVE_CPU_IPL(old_ipl, INTERRUPT_PRIORITY_MAX);

    tx_pending = true;
    tx_success = false;
    tx_ctrl = get_transmission_controller();

    reset_to_zero_tx_timer();
    enable_tx_timer();

    request_tx(tx_ctrl, &frame_to_tx);

    RESTORE_CPU_IPL(old_ipl);

    *output_tx_status = RECANCENTRATE_STATUS_TX_REQUEST_SUCCESSFUL;
}



void read_received_data(
    unsigned int *output_frame_identifier,
    unsigned char *output_frame_data,
    unsigned char *output_frame_length,
    t_recancentrate_status *output_rx_status
)
{
    /* old interrupt priority level */
    int old_ipl;

    if (!rx_data_available) {
        *output_rx_status = RECANCENTRATE_STATUS_NO_RX_DATA_AVAILABLE;
        return;
    }

    SET_AND_SAVE_CPU_IPL(old_ipl, INTERRUPT_PRIORITY_MAX);

    *output_frame_identifier = frame_to_deliver.identifier;
    copy_data(frame_to_deliver.data, frame_to_deliver.length,
        output_frame_data);
    *output_frame_length = frame_to_deliver.length;

    RESTORE_CPU_IPL(old_ipl);

    *output_rx_status = RECANCENTRATE_STATUS_RX_DATA_LOADED;
    rx_data_available = false;
}



bool received_data_is_available(void)
{
    return rx_data_available;
}
```

```
bool recancentrate_tx_carried_out(void)
{
    return tx_success;
}


bool recancentrate_controller_available(void)
{
    return active_controller_available;
}
```

D.2. recancentrate.h

```
/*
 * recancentrate.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */


#ifndef _RECANCENTRATE_H_
#define _RECANCENTRATE_H_

#include "common.h"


typedef enum {
    RECANCENTRATE_STATUS_TX_ALREADY_PENDING,
    RECANCENTRATE_STATUS_TX_REQUEST_SUCCESSFUL,
    RECANCENTRATE_STATUS_NO_RX_DATA_AVAILABLE,
    RECANCENTRATE_STATUS_RX_DATA_LOADED
} t_recancentrate_status;



void init_recancentrate_driver(void);

void request_recancentrate_tx(
    unsigned int frame_identifier,
    unsigned char *frame_data,
    unsigned char frame_length,
    t_recancentrate_status *output_tx_status
);



void read_received_data(
    unsigned int *output_frame_identifier,
    unsigned char *output_frame_data,
```

```
    unsigned char *output_frame_length,
    t_recancentrate_status *output_rx_status
);


bool received_data_is_available(void);

bool recancentrate_tx_carried_out(void);

bool recancentrate_controller_available(void);

#endif /* _RECANCENTRATE_H_ */
```

E. ReCanCentrate hub user constraints file

E.1. cancentrate.ucf

```
    #PACE: Pines normales


# Switch 2 is reset
NET "nRST"  LOC = "E11"  ;

NET "clkOscl"  LOC = "H16"  ;

# Switch 3 is fault injection
NET "injectFault" LOC = "A13";

NET "rx_0"  LOC = "H15"  ;
NET "rx_1"  LOC = "H14"  ;
NET "rx_2"  LOC = "G12"  ;
NET "rehRx_0"  LOC = "G16"  ;           # Bank 5
NET "rehRx_1"  LOC = "H13"  ;

NET "tx_0"  LOC = "G15"  ;
NET "tx_1"  LOC = "G14"  ;
NET "tx_2"  LOC = "F14"  ;
NET "hubTx_0"  LOC = "F15"  ;
NET "hubTx_1"  LOC = "G13"  ;


    #PACE: Pines de depuracion

#NET "dbg_habCRC" LOC = "G12";
#NET "dbg_PetSincro" LOC = "G12";

NET "dbg_brpClk"  LOC = "J14"  ;
NET "dbg_synClkR"  LOC = "E16"  ;
NET "dbg_synROut"  LOC = "E15"  ;
NET "dbg_synClkT"  LOC = "D16"  ;


NET "dbg_stuBitStuffWaited"  LOC = "E14"  ;
NET "dbg_stuValueBitStuffWaited"  LOC = "D15"  ;
NET "dbg_gfmIniErrorFrame"  LOC = "F5"  ;
NET "dbg_gfmLastBitEof"  LOC = "D2"  ;
```

```
NET "dbg_gfmGlobalFrameState<3>"  LOC = "D1"  ;
NET "dbg_gfmGlobalFrameState<2>"  LOC = "F4"  ;
NET "dbg_gfmGlobalFrameState<1>"  LOC = "E2"  ;      # Bank 4
NET "dbg_gfmGlobalFrameState<0>"  LOC = "E1"  ;

NET "dbg_estatErrorFrmGen"  LOC = "F3"  ;


NET "dbg_estatThreshold_0<1>"  LOC = "G5"  ;
NET "dbg_estatThreshold_0<0>"  LOC = "F2"  ;

NET "dbg_estatThreshold_1<1>"  LOC = "G4"  ;
NET "dbg_estatThreshold_1<0>"  LOC = "G3"  ;

#SUPR LIMTBOARD NET "dbg_estatThreshold_2<1>"  LOC = "G2"  ;
#SUPR LIMTBOARD NET "dbg_estatThreshold_2<0>"  LOC = "G1"  ;

NET "dbg_estatHubThreshold_0<1>"    LOC = "G2"  ;
NET "dbg_estatHubThreshold_0<0>"    LOC = "G1"  ;

NET "dbg_estatHubThreshold_1<1>"    LOC = "H4"  ;
NET "dbg_estatHubThreshold_1<0>"    LOC = "H3"  ;



NET "dbg_portType_0"  LOC = "H1"  ;
NET "dbg_portType_1"  LOC = "J1"  ;
#SUPR LIMTBOARD NET "dbg_portType_2"  LOC = "J2"  ;
NET "dbg_hubType_0"  LOC = "J2"  ;
NET "dbg_hubType_1"  LOC = "J3"  ;


NET "dbg_estatBICManager_0<2>"  LOC = "K1"  ;
NET "dbg_estatBICManager_0<1>"  LOC = "L2"  ;
NET "dbg_estatBICManager_0<0>"  LOC = "K5"  ;

NET "dbg_estatBICManager_1<2>"  LOC = "L3"  ;
NET "dbg_estatBICManager_1<1>"  LOC = "M1"  ;
NET "dbg_estatBICManager_1<0>"  LOC = "M2"  ;

#SUPR LIMTBOARD NET "dbg_estatBICManager_2<2>"  LOC = "L4"  ;
#SUPR LIMTBOARD NET "dbg_estatBICManager_2<1>"  LOC = "N1"  ;
#SUPR LIMTBOARD NET "dbg_estatBICManager_2<0>"  LOC = "M3"  ;

NET "dbg_estatHubBICManager_0<2>"  LOC = "L4"  ;
NET "dbg_estatHubBICManager_0<1>"  LOC = "N1"  ;
NET "dbg_estatHubBICManager_0<0>"  LOC = "M3"  ;

NET "dbg_estatHubBICManager_1<2>"  LOC = "N2"  ;
NET "dbg_estatHubBICManager_1<1>"  LOC = "L5"  ;
NET "dbg_estatHubBICManager_1<0>"  LOC = "P1"  ;


NET "dbg_bimBICValue_0<4>"  LOC = "M4"  ;
```

```
NET "dbg_bimBICValue_0<3>"  LOC = "N3"  ;
NET "dbg_bimBICValue_0<2>"  LOC = "P2"  ;
NET "dbg_bimBICValue_0<1>"  LOC = "R1"  ;        # Bank 3
NET "dbg_bimBICValue_0<0>"  LOC = "M10"  ;

NET "dbg_bimBICValue_1<4>"  LOC = "P11"  ;
NET "dbg_bimBICValue_1<3>"  LOC = "T12"  ;
NET "dbg_bimBICValue_1<2>"  LOC = "R12"  ;
NET "dbg_bimBICValue_1<1>"  LOC = "T13"  ;
NET "dbg_bimBICValue_1<0>"  LOC = "P12"  ;

#SUPR LIMTBOARD NET "dbg_bimBICValue_2<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_bimBICValue_2<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_bimBICValue_2<0>"  LOC = ""  ;

NET "dbg_hbmBICValue_0<6>"  LOC = "N15"  ;
NET "dbg_hbmBICValue_0<5>"  LOC = "M15"  ;
NET "dbg_hbmBICValue_0<4>"  LOC = "L15"  ;
NET "dbg_hbmBICValue_0<3>"  LOC = "K15"  ;
NET "dbg_hbmBICValue_0<2>"  LOC = "K16"  ;
NET "dbg_hbmBICValue_0<1>"  LOC = "J13"  ;
#NET "dbg_hbmBICValue_0<0>"  LOC = "J14"  ;

NET "dbg_hbmBICValue_1<6>"  LOC = "J16"  ;
#SUPR LIMTBOARD NET "dbg_hbmBICValue_1<5>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_hbmBICValue_1<4>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_hbmBICValue_1<3>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_hbmBICValue_1<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_hbmBICValue_1<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_hbmBICValue_1<0>"  LOC = ""  ;



#SUPR LIMTBOARD NET "dbg_namNACKValue_0<2>"  LOC = ""  ;        # Bank 2
#SUPR LIMTBOARD NET "dbg_namNACKValue_0<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_namNACKValue_0<0>"  LOC = ""  ;

#SUPR LIMTBOARD NET "dbg_namNACKValue_1<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_namNACKValue_1<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_namNACKValue_1<0>"  LOC = ""  ;

#SUPR LIMTBOARD NET "dbg_namNACKValue_2<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_namNACKValue_2<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_namNACKValue_2<0>"  LOC = ""  ;

#SUPR LIMTBOARD NET "dbg_namNACKValue_3<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_namNACKValue_3<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_namNACKValue_3<0>"  LOC = ""  ;


#SUPR LIMTBOARD NET "dbg_dbmDBCValue_0<4>"  LOC = ""  ;      # Bank 1
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_0<3>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_0<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_0<1>"  LOC = ""  ;
```

```
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_0<0>"  LOC = ""  ;

#SUPR LIMTBOARD NET "dbg_dbmDBCValue_1<4>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_1<3>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_1<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_1<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_1<0>"  LOC = ""  ;

#SUPR LIMTBOARD NET "dbg_dbmDBCValue_2<4>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_2<3>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_2<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_2<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_2<0>"  LOC = ""  ;      # Bank 0

#SUPR LIMTBOARD NET "dbg_dbmDBCValue_3<4>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_3<3>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_3<2>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_3<1>"  LOC = ""  ;
#SUPR LIMTBOARD NET "dbg_dbmDBCValue_3<0>"  LOC = ""  ;


#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints

#PACE: End of Constraints generated by PACE
```

F. Source code for fault injection

F.1. Files to inject controller crashes

F.1.1. crash_controller.h

```
/*
 * interrupts.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */


#ifndef _CRASH_CONTROLLER_H_
#define _CRASH_CONTROLLER_H_


void enable_button_interrupts(void);


#endif /* _CRASH_CONTROLLER_H_ */
```

F.1.2. crash_controller.c

```
/*
 * crash_controller.c
 *
 * Written by David Gessner <davidges@gmail.com>
 *
 *
 * Used to simulate a controller crash by pressing one of the buttons on the
 * dsPICDEM board.
 */

#include "../can_controller.h"
#include "../interrupts.h"


extern volatile struct can_controller ctrl1, ctrl2;


/* We use the IC1 and IC2 interrupts as the crash controller 1 and crash
 * controller 2 interrupts */


void enable_button_interrupts(void)
{
```

```
    IPC0bits.IC1IP = INTERRUPT_PRIORITY_MAX;
    IPC1bits.IC2IP = INTERRUPT_PRIORITY_MAX;

    IEC0bits.IC1IE = 1;
    IEC0bits.IC2IE = 1;
}



/* Invoked when button S1 is pressed */
void __attribute__((__interrupt__, no_auto_psv)) _IC1Interrupt(void)
{
    shutdown(&ctrl1);

    /* Acknowledge interrupt */
    IFS0bits.IC1IF = 0;
}



/* Invoked when button S2 is pressed */
void __attribute__((__interrupt__, no_auto_psv)) _IC2Interrupt(void)
{
    shutdown(&ctrl2);

    /* Acknowledge interrupt */
    IFS0bits.IC2IF = 0;
}
```

F.2. Fault-injection modules

F.2.1. downlinkFaultInjectionModule.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.defInteger.all;
use work.defStates.all;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity faultInjectionModule is
    port
    (
        -- System reset
        sysReset:    in std_logic;

        -- Clocks
```

```
        synClkR:     in std_logic;
        synClkT:     in std_logic;

        -- Mapped to switch button 3, which, when pressed, indicates
        -- that fault injection should start
        iomInjectFault: in std_logic;

        -- State of globalFrameMonitorUnit (current state of the
        -- resultant frame)
        gfmGlobalFrameState:     in estadoGlobalFrame;

        -- Values to be injected into the downlinks
        fimTx_0:     out std_logic;
        fimTx_1:     out std_logic;
        fimTx_2:     out std_logic;

        -- Values to be injected into the uplinks
        fimRx_0:     out std_logic;
        fimRx_1:     out std_logic;
        fimRx_2:     out std_logic;

        -- Values to be injected into the sublinks
        fimHubTx_0: out std_logic;
        fimHubTx_1: out std_logic;

        -- Booleans which indicate whether a fault should be injected
        -- or not through the given downlink
        fimInjTx_0: out std_logic;
        fimInjTx_1: out std_logic;
        fimInjTx_2: out std_logic;

        -- Booleans which indicate whether a fault should be injected
        -- or not through the given uplink
        fimInjRx_0: out std_logic;
        fimInjRx_1: out std_logic;
        fimInjRx_2: out std_logic;

        -- Booleans which indicate whether a fault should be injected
        -- or not through the given sublink
        fimInjHubTx_0:  out std_logic;
        fimInjHubTx_1:  out std_logic
    );
end faultInjectionModule;

architecture Behavioral of faultInjectionModule is

    type estadoFaultInjection is (nonFaulty, preFaulty, faulty, postFaulty);
    signal estatFaultInjection: estadoFaultInjection;


    -- Auxiliary signals

    signal auxFimTx_0: std_logic;
    signal auxFimTx_1: std_logic;
```

```
    signal auxFimTx_2: std_logic;

    signal auxFimRx_0: std_logic;
    signal auxFimRx_1: std_logic;
    signal auxFimRx_2: std_logic;

    signal auxFimHubTx_0: std_logic;
    signal auxFimHubTx_1: std_logic;

    signal auxFimInjecting: std_logic;

    -- Previous value of iomInjectFault
    signal antInjectFault: std_logic;
begin

    -- Map auxiliary signals to outputs
    fimTx_0 <= auxFimTx_0;
    fimTx_1 <= auxFimTx_1;
    fimTx_2 <= auxFimTx_2;

    fimRx_0 <= auxFimRx_0;
    fimRx_1 <= auxFimRx_1;
    fimRx_2 <= auxFimRx_2;

    fimHubTx_0 <= auxFimHubTx_0;
    fimHubTx_1 <= auxFimHubTx_1;


     -- Detects when switch button 3 is pressed
    process (synClkR, sysReset) is
    begin
        if sysReset = '1' then
            antInjectFault <= iomInjectFault;
            auxFimInjecting <= '0';

        elsif synClkR'event and synClkR = '1' then
            if iomInjectFault = '0' and antInjectFault = '1' then
                auxFimInjecting <= not auxFimInjecting;
            end if;
            antInjectFault <= iomInjectFault;
        end if;
    end process;


    -- State machine that controls fault injection
    process (synClkT, sysReset) is
    begin
        if sysReset = '1' then
            estatFaultInjection <= nonFaulty;

            -- Initialize signals to be injected into downlinks
            auxFimTx_0 <= '1';
            auxFimTx_1 <= '1';
            auxFimTx_2 <= '1';
```

```
        -- Initialize signals to be injected into uplinks
        auxFimRx_0 <= '1';
        auxFimRx_1 <= '1';
        auxFimRx_2 <= '1';
        -- Initialize signals to be injected into sublinks
        auxFimHubTx_0 <= '1';
        auxFimHubTx_1 <= '1';

        -- Downlink fault injection disabled
        fimInjTx_0 <= '0';
        fimInjTx_1 <= '0';
        fimInjTx_2 <= '0';
        -- Uplink fault injection disabled
        fimInjRx_0 <= '0';
        fimInjRx_1 <= '0';
        fimInjRx_2 <= '0';
        -- Sublink fault injection disabled
        fimInjHubTx_0 <= '0';
        fimInjHubTx_1 <= '0';

    elsif synClkT'event and synClkT = '1' then
        case (estatFaultInjection) is
            when nonFaulty =>
                -- Downlink fault injection disabled
                fimInjTx_0 <= '0';
                fimInjTx_1 <= '0';
                fimInjTx_2 <= '0';
                -- Uplink fault injection disabled
                fimInjRx_0 <= '0';
                fimInjRx_1 <= '0';
                fimInjRx_2 <= '0';
                -- Sublink fault injection disabled
                fimInjHubTx_0 <= '0';
                fimInjHubTx_1 <= '0';

                -- If switch button pressed
                if auxFimInjecting = '1' then
                    -- Start fault injection
                    estatFaultInjection <= preFaulty;
                end if;

            when preFaulty =>
                -- Posible values for gfmGlobalFrameState:
                -- idle, idField, rtrField, resField,
                -- dlcField, dataField, crcField,
                -- crcDelimField, ackSlotField,
                -- ackDelimField, eofField, interField,
                -- errorFlag, errorDelimiter,
                -- freeState1, freeState2

                -- Inject fault during EOF field
                if gfmGlobalFrameState = eofField then
                    estatFaultInjection <= faulty;
                end if;
```

```
                when faulty =>

                    -- 1 enables faults in downlink Tx_0,
                    -- 0 disables faults
                    fimInjTx_0 <= '1';
                    -- ... inject stuck-at-recessive
                    -- auxFimTx_0 <= '1';
                    -- ... inject stuck-at-dominant
                    auxFimTx_0 <= '0';
                    -- ... inject bit-flipping
                    -- auxFimTx_0 <= not auxFimTx_0;

                    -- Detects when switch button 3 is
                    -- pressed
                    if auxFimInjecting = '0' then
                        -- Begin stop of fault injection
                        estatFaultInjection <= postFaulty;
                    end if;

                when postFaulty =>
                    -- Stop fault injection during EOF (if
                    -- we'd stop fault injection at any
                    -- arbitrary moment, the
                    -- downlink/sublink/uplink could become
                    -- non-faulty in the middle of a frame,
                    -- and the remaining bits of that frame
                    -- could be interpreted as bit flippings)
                    if gfmGlobalFrameState = eofField then
                        estatFaultInjection <= nonFaulty;
                    end if;

                when others =>
                    estatFaultInjection <= nonFaulty;

            end case;
        end if;
    end process;

end Behavioral;
```

F.2.2. uplinkFaultInjectionModule.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.defInteger.all;
use work.defStates.all;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
```

```
--library UNISIM;
--use UNISIM.VComponents.all;

entity faultInjectionModule is
    port
    (
        -- System reset
        sysReset:    in std_logic;

        -- Clocks
        synClkR:     in std_logic;
        synClkT:     in std_logic;

        -- Mapped to switch button 3, which, when pressed, indicates
        -- that fault injection should start
        iomInjectFault: in std_logic;

        -- State of globalFrameMonitorUnit (current state of the
        -- resultant frame)
        gfmGlobalFrameState:     in estadoGlobalFrame;

        -- Values to be injected into the downlinks
        fimTx_0:     out std_logic;
        fimTx_1:     out std_logic;
        fimTx_2:     out std_logic;

        -- Values to be injected into the uplinks
        fimRx_0:     out std_logic;
        fimRx_1:     out std_logic;
        fimRx_2:     out std_logic;

        -- Values to be injected into the sublinks
        fimHubTx_0: out std_logic;
        fimHubTx_1: out std_logic;

        -- Booleans which indicate whether a fault should be injected
        -- or not through the given downlink
        fimInjTx_0: out std_logic;
        fimInjTx_1: out std_logic;
        fimInjTx_2: out std_logic;

        -- Booleans which indicate whether a fault should be injected
        -- or not through the given uplink
        fimInjRx_0: out std_logic;
        fimInjRx_1: out std_logic;
        fimInjRx_2: out std_logic;

        -- Booleans which indicate whether a fault should be injected
        -- or not through the given sublink
        fimInjHubTx_0:  out std_logic;
        fimInjHubTx_1:  out std_logic
    );
end faultInjectionModule;
```

```
architecture Behavioral of faultInjectionModule is

    type estadoFaultInjection is (nonFaulty, preFaulty, faulty, postFaulty);
    signal estatFaultInjection: estadoFaultInjection;


    -- Auxiliary signals

    signal auxFimTx_0: std_logic;
    signal auxFimTx_1: std_logic;
    signal auxFimTx_2: std_logic;

    signal auxFimRx_0: std_logic;
    signal auxFimRx_1: std_logic;
    signal auxFimRx_2: std_logic;

    signal auxFimHubTx_0: std_logic;
    signal auxFimHubTx_1: std_logic;

    signal auxFimInjecting: std_logic;

    -- Previous value of iomInjectFault
    signal antInjectFault: std_logic;
begin

    -- Map auxiliary signals to outputs
    fimTx_0 <= auxFimTx_0;
    fimTx_1 <= auxFimTx_1;
    fimTx_2 <= auxFimTx_2;

    fimRx_0 <= auxFimRx_0;
    fimRx_1 <= auxFimRx_1;
    fimRx_2 <= auxFimRx_2;

    fimHubTx_0 <= auxFimHubTx_0;
    fimHubTx_1 <= auxFimHubTx_1;


     -- Detects when switch button 3 is pressed
    process (synClkR, sysReset) is
    begin
        if sysReset = '1' then
            antInjectFault <= iomInjectFault;
            auxFimInjecting <= '0';

        elsif synClkR'event and synClkR = '1' then
            if iomInjectFault = '0' and antInjectFault = '1' then
                auxFimInjecting <= not auxFimInjecting;
            end if;
            antInjectFault <= iomInjectFault;
        end if;
    end process;
```

```
-- State machine that controls fault injection
process (synClkT, sysReset) is
begin
    if sysReset = '1' then
        estatFaultInjection <= nonFaulty;

        -- Initialize signals to be injected into downlinks
        auxFimTx_0 <= '1';
        auxFimTx_1 <= '1';
        auxFimTx_2 <= '1';
        -- Initialize signals to be injected into uplinks
        auxFimRx_0 <= '1';
        auxFimRx_1 <= '1';
        auxFimRx_2 <= '1';
        -- Initialize signals to be injected into sublinks
        auxFimHubTx_0 <= '1';
        auxFimHubTx_1 <= '1';

        -- Downlink fault injection disabled
        fimInjTx_0 <= '0';
        fimInjTx_1 <= '0';
        fimInjTx_2 <= '0';
        -- Uplink fault injection disabled
        fimInjRx_0 <= '0';
        fimInjRx_1 <= '0';
        fimInjRx_2 <= '0';
        -- Sublink fault injection disabled
        fimInjHubTx_0 <= '0';
        fimInjHubTx_1 <= '0';

    elsif synClkT'event and synClkT = '1' then
        case (estatFaultInjection) is
            when nonFaulty =>
                -- Downlink fault injection disabled
                fimInjTx_0 <= '0';
                fimInjTx_1 <= '0';
                fimInjTx_2 <= '0';
                -- Uplink fault injection disabled
                fimInjRx_0 <= '0';
                fimInjRx_1 <= '0';
                fimInjRx_2 <= '0';
                -- Sublink fault injection disabled
                fimInjHubTx_0 <= '0';
                fimInjHubTx_1 <= '0';

                -- If switch button pressed
                if auxFimInjecting = '1' then
                    -- Start fault injection
                    estatFaultInjection <= preFaulty;
                end if;

            when preFaulty =>
                -- Posible values for gfmGlobalFrameState:
                -- idle, idField, rtrField, resField,
```

```
                    -- dlcField, dataField, crcField,
                    -- crcDelimField, ackSlotField,
                    -- ackDelimField, eofField, interField,
                    -- errorFlag, errorDelimiter,
                    -- freeState1, freeState2

                    -- Inject fault during EOF field
                    if gfmGlobalFrameState = eofField then
                        estatFaultInjection <= faulty;
                    end if;


                when faulty =>

                    -- 1 enables faults in uplink Rx_0,
                    -- 0 disables faults
                    fimInjRx_0 <= '1';
                    -- ... inject stuck-at-recessive
                    auxFimRx_0 <= '1';
                    -- ... inject stuck-at-dominant
                    --auxFimRx_0 <= '0';
                    -- ... inject bit-flipping
                    -- auxFimRx_0 <= not auxFimTx_0;

                    -- Detects when switch button 3 is
                    -- pressed
                    if auxFimInjecting = '0' then
                        -- Begin stop of fault injection
                        estatFaultInjection <= postFaulty;
                    end if;

                when postFaulty =>
                    -- Stop fault injection during EOF (if
                    -- we'd stop fault injection at any
                    -- arbitrary moment, the
                    -- downlink/sublink/uplink could become
                    -- non-faulty in the middle of a frame,
                    -- and the remaining bits of that frame
                    -- could be interpreted as bit flippings)
                    if gfmGlobalFrameState = eofField then
                        estatFaultInjection <= nonFaulty;
                    end if;

                when others =>
                    estatFaultInjection <= nonFaulty;

            end case;
        end if;
    end process;

end Behavioral;
```

F.2.3. ReCanCentrate.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.defInteger.all;
use work.defStates.all;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity ReCanCentrate is
    port
    (
        -- ======== Debugging ==========
        dbg_brpClk:          out std_logic;

        dbg_habCRC:          out std_logic;
        dbg_synClkR:             out std_logic;
        dbg_synROut:             out std_logic;
        dbg_synClkT:             out std_logic;

        dbg_stuBitStuffWaited:       out std_logic;
        dbg_stuValueBitStuffWaited: out std_logic;
        dbg_gfmIniErrorFrame:        out std_logic;
        dbg_gfmLastBitEof:       out std_logic;
        dbg_gfmGlobalFrameState:     out std_logic_vector(3 downto 0);
        dbg_gfmErrorCRC:         out std_logic;

        dbg_estatErrorFrmGen:        out std_logic;

        dbg_estatThreshold_0:        out std_logic_vector (1 downto 0);
        dbg_estatThreshold_1:        out std_logic_vector (1 downto 0);
        dbg_estatHubThreshold_0:     out std_logic_vector (1 downto 0);
        dbg_estatHubThreshold_1:     out std_logic_vector (1 downto 0);

        dbg_portType_0:          out std_logic;
        dbg_portType_1:          out std_logic;
        dbg_hubType_0:           out std_logic;
        dbg_hubType_1:           out std_logic;

        dbg_estatBICManager_0:       out std_logic_vector (2 downto 0);
        dbg_estatBICManager_1:       out std_logic_vector (2 downto 0);
        dbg_estatHubBicManager_0:        out std_logic_vector (2 downto 0);
        dbg_estatHubBicManager_1:        out std_logic_vector (2 downto 0);

        dbg_bimBICValue_0:           out std_logic_vector (4 downto 0);
        dbg_bimBICValue_1:           out std_logic_vector (4 downto 0);
        dbg_hbmBICValue_0:           out std_logic_vector (6 downto 0);
        dbg_hbmBICValue_1:           out std_logic_vector (6 downto 0);

        -- ======================
```

```
        -- External reset signal
        nRST:           in std_logic;

        -- Oscillator clock
        clkOscl:        in std_logic;

        -- Mapped to switch button 3
        injectFault:    in std_logic;

        -- Uplinks
        rx_0:       in std_logic;
        rx_1:       in std_logic;
        rx_2:       in std_logic;

        -- Incoming sublinks
        rehRx_0:    in std_logic;
        rehRx_1:    in std_logic;


        -- Downlinks
        tx_0:       out std_logic;
        tx_1:       out std_logic;
        tx_2:       out std_logic;

        -- Outgoing sublinks
        hubTx_0:    out std_logic;
        hubTx_1:    out std_logic
    );
end ReCanCentrate;

architecture Behavioral of ReCanCentrate is

    component faultInjectionModule is
        port
        (
            -- System reset
            sysReset:   in std_logic;

            -- Clocks
            synClkR:    in std_logic;
            synClkT:    in std_logic;

            -- Indicates whether faults should be injected or not
            iomInjectFault: in std_logic;

            -- State of globalFrameMonitorUnit (current state of
            -- the resultant frame)
            gfmGlobalFrameState:    in estadoGlobalFrame;

            -- Values to be injected into the downlinks
            fimTx_0:    out std_logic;
            fimTx_1:    out std_logic;
            fimTx_2:    out std_logic;
```

```
        -- Values to be injected into the uplinks
        fimRx_0:     out std_logic;
        fimRx_1:     out std_logic;
        fimRx_2:     out std_logic;

        -- Values to be injected into the sublinks
        fimHubTx_0: out std_logic;
        fimHubTx_1: out std_logic;

        -- Booleans which indicate whether a fault should be
        -- injected or not through the given downlink
        fimInjTx_0: out std_logic;
        fimInjTx_1: out std_logic;
        fimInjTx_2: out std_logic;

        -- Booleans which indicate whether a fault should be
        -- injected or not through the given uplink
        fimInjRx_0: out std_logic;
        fimInjRx_1: out std_logic;
        fimInjRx_2: out std_logic;

        -- Booleans which indicate whether a fault should be
        -- injected or not through the given sublink
        fimInjHubTx_0:  out std_logic;
        fimInjHubTx_1:  out std_logic
    );
end component;



component bufGlobalPrimaryClock is
    port
    (
        clk:     in std_logic;

        bufClk: out std_logic
    );
end component;

component couplerRepModule is
    port
    (
        cplCoupledSignal:   in std_logic;
        iomRepContri_0:     in std_logic;
        iomRepContri_1:     in std_logic;

        htmEnaDisRep_0:     in std_logic;
        thmEnaDisRep_1:     in std_logic;

        crpCoupledSignal:   out std_logic
    );
end component;
```

```
component couplerModule is
    port
    (
        iomPortContri_0:    in std_logic;
        iomPortContri_1:    in std_logic;
        iomPortContri_2:    in std_logic;

        thmEnaDis_0:        in std_logic;
        thmEnaDis_1:        in std_logic;
        thmEnaDis_2:        in std_logic;

        efgTxSignal:        in std_logic;

        cplCoupledSignal:   out std_logic
    );
end component;


component faultTreatmentModule is
port (
        -- ======== Debugging ==========
        dbg_brpClk:     out std_logic;

        dbg_habCRC:     out std_logic;
        dbg_synClkR:        out std_logic;
        dbg_synROut:        out std_logic;
        dbg_synClkT:        out std_logic;

        dbg_stuBitStuffWaited:  out std_logic;
        dbg_stuValueBitStuffWaited: out std_logic;
        dbg_gfmIniErrorFrame:       out std_logic;
        dbg_gfmLastBitEof:      out std_logic;
        dbg_gfmGlobalFrameState:    out std_logic_vector(3 downto 0);
        dbg_gfmErrorCRC:        out std_logic;

        dbg_estatErrorFrmGen:       out std_logic;

        dbg_estatThreshold_0:       out std_logic_vector (1 downto 0);
        dbg_estatThreshold_1:       out std_logic_vector (1 downto 0);
        dbg_estatHubThreshold_0:    out std_logic_vector (1 downto 0);
        dbg_estatHubThreshold_1:    out std_logic_vector (1 downto 0);

        dbg_portType_0:         out std_logic;
        dbg_portType_1:         out std_logic;
        dbg_HubType_0:          out std_logic;
        dbg_HubType_1:          out std_logic;

        dbg_estatBICManager_0:      out std_logic_vector (2 downto 0);
        dbg_estatBICManager_1:      out std_logic_vector (2 downto 0);
        dbg_estatHubBicManager_0:   out std_logic_vector (2 downto 0);
        dbg_estatHubBicManager_1:   out std_logic_vector (2 downto 0);

        dbg_bimBICValue_0:      out std_logic_vector (4 downto 0);
        dbg_bimBICValue_1:      out std_logic_vector (4 downto 0);
```

```
        dbg_hbmBICValue_0:      out std_logic_vector (6 downto 0);
        dbg_hbmBICValue_1:      out std_logic_vector (6 downto 0);

        -- ======================
        -- Reset, oscillator, and resultant signal
        sysReset:    in std_logic;
        clk:         in std_logic;
        crpCoupledSignal: in std_logic;

        -- Node contributions
        iomPortContri_0:    in std_logic;
        iomPortContri_1:    in std_logic;
        iomPortContri_2:    in std_logic;

        -- Contributions of the other hub
        iomHubContri_0:     in  std_logic;
        iomHubContri_1:     in  std_logic;

        -- Bit-flipping counter threshold
        thresholdBIC:       in std_logic_vector (4 downto 0);
        -- Non ACKnowledge counter threshold
        thresholdNACK:      in std_logic_vector (2 downto 0);
        -- Dominant bit counter threshold
        thresholdDBC:       in std_logic_vector (4 downto 0);

        -- Bit rate configuration
        brp:         in std_logic_vector (5 downto 0);
        tsegment1:   in  std_logic_vector (5 downto 0);
        tsegment2:   in  std_logic_vector (2 downto 0);
        sjw:         in  std_logic_vector (1 downto 0);
        -- If 0, resynchronization with 'r' to 'd' edge; if 1,
        -- resynchronization with both 'r' to 'd' and 'd' to 'r'
        sync:        in  std_logic;
        -- If 0, sample once; if 1, sample 3 times
        sam:         in  std_logic;

        -- Signals to enable or disable uplinks
        thmEnaDis_0:    out std_logic;
        thmEnaDis_1:    out std_logic;
        thmEnaDis_2:    out std_logic;

        -- Signals to enable or disable sublinks
        htmEnaDis_0:    out std_logic;
        htmEnaDis_1:    out std_logic;

        -- The hub's transmit contribution
        efgTxSignal:    out std_logic;

        -- State of globalFrameMonitorUnit (current state of
        -- the resultant frame)
        gfmGlobalFrameState:    out estadoGlobalFrame
    );
end component;
```

```
component biestableD is
port(
        reset:      in std_logic;
        clk:        in std_logic;
        entrada:    in std_logic;
        salida:     out std_logic
    );
end component;


signal sysReset: std_logic;
signal clk: std_logic;
signal auxCplCoupledSignal: std_logic;
signal auxCrpCoupledSignal: std_logic;

signal auxHrvTx_0: std_logic;
signal auxHrvTx_1: std_logic;

signal auxEfgTxSignal: std_logic;

signal auxThmEnaDis_0: std_logic;
signal auxthmEnaDis_1: std_logic;
signal auxThmEnaDis_2: std_logic;
signal auxThmEnaDis_3: std_logic;

signal auxHtmEnaDis_0: std_logic;
signal auxHtmEnaDis_1: std_logic;

signal auxFimTx_0: std_logic;
signal auxFimTx_1: std_logic;
signal auxFimTx_2: std_logic;
signal auxFimRx_0: std_logic;
signal auxFimRx_1: std_logic;
signal auxFimRx_2: std_logic;
signal auxFimHubTx_0: std_logic;
signal auxFimHubTx_1: std_logic;

signal auxFimInjTx_0: std_logic;
signal auxFimInjTx_1: std_logic;
signal auxFimInjTx_2: std_logic;

signal auxFimInjRx_0: std_logic;
signal auxFimInjRx_1: std_logic;
signal auxFimInjRx_2: std_logic;

signal auxFimInjHubTx_0: std_logic;
signal auxFimInjHubTx_1: std_logic;

signal auxRx_0: std_logic;
signal auxRx_1: std_logic;
signal auxRx_2: std_logic;

signal auxauxRx_0: std_logic;
signal auxauxRx_1: std_logic;
```

```
    signal auxauxRx_2: std_logic;

    signal auxRehRx_0: std_logic;
    signal auxRehRx_1: std_logic;

    signal auxSynClkR: std_logic;
    signal auxSynClkT: std_logic;

    signal auxGfmGlobalFrameState: estadoGlobalFrame;

begin
    -- Mapping to clocks for debugging
    dbg_synClkR <= auxSynClkR;
    dbg_synClkT <= auxSynClkT;

    -- Calculate system reset
    sysReset <= not nRST;

    -- Assign oscillator clock to a global clock path
    bufClkOscl: bufGlobalPrimaryClock port map (clkOscl, clk);

    -- Broadcast coupled signal or fault to be injected to downlinks
    with auxFimInjTx_0 select
        tx_0 <=
            (auxCrpCoupledSignal or not nRST)   when '0',
             auxFimTx_0                         when others;

    with auxFimInjTx_1 select
        tx_1 <=
            (auxCrpCoupledSignal or not nRST)   when '0',
             auxFimTx_1                         when others;

    with auxFimInjTx_2 select
        tx_2 <=
            (auxCrpCoupledSignal or not nRST)   when '0',
             auxFimTx_2                         when others;


    -- Assign uplink signals or fault to be injected to uplinks
    with auxFimInjRx_0 select
        auxauxRx_0 <=
            (auxRx_0)                           when '0',
            auxFimRx_0                          when others;

    with auxFimInjRx_1 select
        auxauxRx_1 <=
            (auxRx_1)                           when '0',
            auxFimRx_1                          when others;

    with auxFimInjRx_2 select
        auxauxRx_2 <=
            (auxRx_2)                           when '0',
            auxFimRx_2                          when others;
```

```
-- Assign hub contribution or fault to be injected to outgoing sublinks
with auxFimInjHubTx_0 select
    hubTx_0 <=
        (auxCplCoupledSignal or not nRST)   when '0',
         auxFimHubTx_0                      when others;

with auxFimInjHubTx_1 select
    hubTx_1 <=
        (auxCplCoupledSignal or not nRST)   when '0',
         auxFimHubTx_1                      when others;

biestableRx_0: biestableD port map (sysReset, clk, rx_0, auxRx_0);
biestableRx_1: biestableD port map (sysReset, clk, rx_1, auxRx_1);
biestableRx_2: biestableD port map (sysReset, clk, rx_2, auxRx_2);

biestablerehRx_0: biestableD port map (sysReset, clk, rehRx_0, auxRehRx_0);
biestablerehRx_1: biestableD port map (sysReset, clk, rehRx_1, auxRehRx_1);

faultInjectionModuleUnit:
    faultInjectionModule
        port map (
            sysReset => sysReset,

            synClkR => auxSynClkR,
            synClkT => auxSynClkT,

            iomInjectFault => injectFault,

            gfmGlobalFrameState => auxGfmGlobalFrameState,

            fimTx_0 => auxFimTx_0,
            fimTx_1 => auxFimTx_1,
            fimTx_2 => auxFimTx_2,

            fimRx_0 => auxFimRx_0,
            fimRx_1 => auxFimRx_1,
            fimRx_2 => auxFimRx_2,

            fimHubTx_0 => auxFimHubTx_0,
            fimHubTx_1 => auxFimHubTx_1,

            fimInjTx_0 => auxFimInjTx_0,
            fimInjTx_1 => auxFimInjTx_1,
            fimInjTx_2 => auxFimInjTx_2,

            fimInjRx_0 => auxFimInjRx_0,
            fimInjRx_1 => auxFimInjRx_1,
            fimInjRx_2 => auxFimInjRx_2,

            fimInjHubTx_0 => auxFimInjHubTx_0,
            fimInjHubTx_1 => auxFimInjHubTx_1
        );
```

```
-- Coupler replica module
couplerRepModuleUnit:
    couplerRepModule
        port map (
            cplCoupledSignal => auxCplCoupledSignal,
            iomRepContri_0   => auxRehRx_0,
            iomRepContri_1   => auxRehRx_1,

            htmEnaDisRep_0   => auxHtmEnaDis_0,
            thmEnaDisRep_1   => auxHtmEnaDis_1,

            crpCoupledSignal => auxCrpCoupledSignal
        );


-- Coupler module
couplerModuluUnit:
    couplerModule
        port map (
            iomPortContri_0 => auxauxRx_0,
            iomPortContri_1 => auxauxRx_1,
            iomPortContri_2 => auxauxRx_2,

            thmEnaDis_0 => auxThmEnaDis_0,
            thmEnaDis_1 => auxThmEnaDis_1,
            thmEnaDis_2 => auxThmEnaDis_2,

            efgTxSignal => auxEfgTxSignal,

            cplCoupledSignal => auxCplCoupledSignal
        );


-- Fault treatment module
faultTreatmentModuleUnit:
    faultTreatmentModule
        port map (
            -- ======== Debugging ==========
            dbg_brpClk =>   dbg_brpClk,
            dbg_habCRC => dbg_habCRC,

            dbg_synClkR =>  auxSynClkR,
            dbg_synROut =>  dbg_synROut,
            dbg_synClkT =>  auxSynClkT,

            dbg_stuBitStuffWaited     => dbg_stuBitStuffWaited,
            dbg_stuValueBitStuffWaited =>   dbg_stuValueBitStuffWaited,
            dbg_gfmIniErrorFrame      => dbg_gfmIniErrorFrame,
            dbg_gfmLastBitEof     => dbg_gfmLastBitEof,
            dbg_gfmGlobalFrameState  => dbg_gfmGlobalFrameState,
            dbg_gfmErrorCRC       => dbg_gfmErrorCRC,

            dbg_estatErrorFrmGen      => dbg_estatErrorFrmGen,
```

```
dbg_estatThreshold_0      => dbg_estatThreshold_0,
dbg_estatThreshold_1      => dbg_estatThreshold_1,
dbg_estatHubThreshold_0   => dbg_estatHubThreshold_0,
dbg_estatHubThreshold_1   => dbg_estatHubThreshold_1,

dbg_portType_0 =>    dbg_portType_0,
dbg_portType_1 =>    dbg_portType_1,
dbg_hubType_0 =>     dbg_HubType_0,
dbg_hubType_1 =>     dbg_HubType_1,

dbg_estatBICManager_0 =>     dbg_estatBICManager_0,
dbg_estatBICManager_1 =>     dbg_estatBICManager_1,
dbg_estatHubBicManager_0 => dbg_estatHubBicManager_0,
dbg_estatHubBicManager_1 => dbg_estatHubBicManager_1,

dbg_bimBICValue_0 =>     dbg_bimBICValue_0,
dbg_bimBICValue_1 =>     dbg_bimBICValue_1,
dbg_hbmBICValue_0 =>     dbg_hbmBICValue_0,
dbg_hbmBICValue_1 =>     dbg_hbmBICValue_1,

-- ======================
-- Reset, oscillator, and resultant signal
sysReset => sysReset,
clk   => clk,
crpCoupledSignal => auxCrpCoupledSignal,

-- Node contributions
iomPortContri_0 =>  auxauxRx_0,
iomPortContri_1 =>  auxauxRx_1,
iomPortContri_2 =>  auxauxRx_2,

-- Contributions of the other hub
iomHubContri_0 =>   auxRehRx_0,
iomHubContri_1 =>   auxRehRx_1,

-- Bit-flipping counter threshold
thresholdBIC   =>     "01000",
-- Non ACKnowledge counter threshold
thresholdNACK =>     "111",
-- Dominant bit counter threshold
thresholdDBC   =>     "11000",

-- Bit rate configuration

-- baud rate prescaler
--         2 * (brp + 1)
-- TQ = ----------------
--              clk
brp => "000000",

-- number of TQ for segment 1 (includes
-- propagation segment) = tsegment1 + 1
tsegment1 => "000100",
```

```
            -- number of TQ for segment 2 = tsegment2 + 1
            tsegment2 => "001",

            -- Nominal Bit Time
            -- NBT = (1 + (tsegment1 + 1) + (tsegment2 + 1)) * TQ

            -- number of TQ for synch. jump width = sjw + 1
            sjw => "00",

            -- If sync = 0, only resynchronize with down edge
            -- If sync = 1, resynchronize with down and up edge
            sync => '0',

            -- If sam = 0, sample 1 time,
            -- If sam = 1, sample 3 times
            sam => '0',

            thmEnaDis_0 =>  auxThmEnaDis_0,
            thmEnaDis_1 =>  auxThmEnaDis_1,
            thmEnaDis_2 =>  auxThmEnaDis_2,

            htmEnaDis_0 =>  auxHtmEnaDis_0,
            htmEnaDis_1 =>  auxHtmEnaDis_1,

            efgTxSignal =>  auxEfgTxSignal,

            gfmGlobalFrameState => auxGfmGlobalFrameState
        );

end Behavioral;
```

G. Source code for the fault-tolerance tests

G.1. transmitter_3led_counter.c

```
/*
 * transmitter_3led_counter.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "../../recancentrate.h"
#include "../../device_config.h"
#include "../../assert.h"
#include "../../common.h"
#include "../../led.h"
#include "../crash_controller.h"
#include "../ft-tests.h"



int main(void)
{
    unsigned int sid = LED_COUNTER_ID;
    t_recancentrate_status tx_status;
    /* Data to transmit */
    unsigned char tx_data;
    /* Number of frames transmitted succesfully */
    unsigned char tx_frames_count = 0;

    enable_button_interrupts();
    init_leds();
    init_recancentrate_driver();

    while (1) {
        /* Count on the 3 least-significant LEDs */
        led_display(tx_frames_count % 8);

        tx_data = tx_frames_count;
        request_recancentrate_tx(sid, &tx_data, 1, &tx_status);

        if (tx_status != RECANCENTRATE_STATUS_TX_REQUEST_SUCCESSFUL) {
            ASSERT(false);
        }
```

```
        while (!recancentrate_tx_carried_out()) {
            if (!recancentrate_controller_available()) {
                ASSERT(false);
            }
            /* else do nothing, wait for transmission to be carried
             * out */
        }
        tx_frames_count++;
    }

    return 0;
}
```

G.2. transmitter_blinking_led.c

```
/*
 * transmitter_blinking_led.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "../../recancentrate.h"
#include "../../device_config.h"
#include "../../assert.h"
#include "../../common.h"
#include "../../led.h"
#include "../crash_controller.h"
#include "../ft-tests.h"




int main(void)
{
    unsigned int sid = BLINKER_ID;
    t_recancentrate_status tx_status;
    /* Data to transmit */
    unsigned char tx_data;
    /* Number of frames transmitted succesfully */
    unsigned char tx_frames_count = 0;

    enable_button_interrupts();
    init_leds();
    init_recancentrate_driver();

    while (1) {
        /* Blink most-significant LED */
        led_display((tx_frames_count % 2) << 3);

        tx_data = tx_frames_count;
        request_recancentrate_tx(sid, &tx_data, 1, &tx_status);
```

```
        if (tx_status != RECANCENTRATE_STATUS_TX_REQUEST_SUCCESSFUL) {
            ASSERT(false);
        }

        while (!recancentrate_tx_carried_out()) {
            if (!recancentrate_controller_available()) {
                ASSERT(false);
            }
            /* else do nothing, wait for transmission to be carried
             * out */
        }
        tx_frames_count++;
    }

    return 0;
}
```

G.3. receiver.c

```
/*
 * receiver.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "../recancentrate.h"
#include "../device_config.h"
#include "../assert.h"
#include "../common.h"
#include "../led.h"
#include "crash_controller.h"
#include "ft-tests.h"



unsigned int rx_sid;
/* Received count */
unsigned char rx_count;
unsigned char rx_data_length;
t_recancentrate_status rx_status;
/* Number of frames received from blinker */
unsigned char blinker_frames_count = 0xAA;
/* Number of frames received from 3led_counter */
unsigned char led_counter_frames_count = 0xAA;

void read_recancentrate_frame(void)
{
    while (!received_data_is_available()) {
        if (!recancentrate_controller_available()) {
            ASSERT(false);
        }
        /* else do nothing, wait for data to be received */
    }
```

```
    read_received_data(&rx_sid, &rx_count, &rx_data_length,
            &rx_status);

    if (rx_status != RECANCENTRATE_STATUS_RX_DATA_LOADED) {
        ASSERT(false);
    }
    ASSERT(rx_data_length == 1);
}


void check_counter_value(
    unsigned char reference_count,
    unsigned char *count_to_check,
    int *duplicated_frames_count,
    int *correct_frames_count,
    int *missed_frames_count,
    int *out_of_sync_frames_count
)
{
    unsigned char old_count;
    unsigned char next_count;
    unsigned char next_next_count;

    old_count = *count_to_check;
    next_count = *count_to_check + 1;
    next_next_count = *count_to_check + 2;

    if (reference_count == old_count) {
        /* Received a duplicate of the previous frame */
        (*duplicated_frames_count)++;
    } else if (reference_count == next_count) {
        /* We received the correct frame */
        (*correct_frames_count)++;
    } else if (reference_count == next_next_count) {
        /* We missed a frame */
        (*missed_frames_count)++;
    } else {
        /* We are completely out of sync */
        (*out_of_sync_frames_count)++;
    }
    /* Sync with reference count (if we received a duplicate frame, the
     * synchronization is unnecessary, but done anyway) */
    (*count_to_check) = reference_count;
}


int led_counter_duplicated_frames = 0;
int blinker_duplicated_frames = 0;

int led_counter_missed_frames = 0;
int blinker_missed_frames = 0;

int led_counter_correct_frames = 0;
```

```
int blinker_correct_frames = 0;

int led_counter_out_of_sync_frames = 0;
int blinker_out_of_sync_frames = 0;

int main(void)
{

    enable_button_interrupts();
    init_leds();
    init_recancentrate_driver();

    while (1) {
        read_recancentrate_frame();
        if (rx_sid == LED_COUNTER_ID) {
            check_counter_value(rx_count,
                    &led_counter_frames_count,
                    &led_counter_duplicated_frames,
                    &led_counter_correct_frames,
                    &led_counter_missed_frames,
                    &led_counter_out_of_sync_frames);
            led_display(rx_count % 8);
        } else if (rx_sid == BLINKER_ID) {
            check_counter_value(rx_count,
                    &blinker_frames_count,
                    &blinker_duplicated_frames,
                    &blinker_correct_frames,
                    &blinker_missed_frames,
                    &blinker_out_of_sync_frames);
            led_display((rx_count % 2) << 3);
        } else {
            /* Unknown ID */
            ASSERT(false);
        }
        ASSERT(led_counter_missed_frames == 0);
        ASSERT(blinker_missed_frames == 0);

        /* Only allow at most a single out of sync frame (which is the
         * frame received for the initial synchronization) */
        ASSERT(led_counter_out_of_sync_frames == 0 ||
            led_counter_out_of_sync_frames == 1);
        ASSERT(blinker_out_of_sync_frames == 0 ||
            blinker_out_of_sync_frames == 1);
    }

    return 0;
}
```

H. Source code for the performance tests

H.1. 8byte_transmitter.c

```
/*
 * 8byte_transmitter.c
 *
 * Transmits 8 bytes of data, which is the maximum amount of data that can be
 * send in a single CAN frame.
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "../recancentrate.h"
#include "../device_config.h"
#include "../assert.h"
#include "../common.h"
#include "../led.h"
#include "crash_controller.h"




int main(void)
{
    unsigned int sid = 1;
    t_recancentrate_status tx_status;
    unsigned char tx_data[8] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08};

    enable_button_interrupts();
    init_leds();
    init_recancentrate_driver();

    led_display(tx_data[0]);
    request_recancentrate_tx(sid, tx_data, 8, &tx_status);

    if (tx_status != RECANCENTRATE_STATUS_TX_REQUEST_SUCCESSFUL) {
        ASSERT(false);
    }

    while (!recancentrate_tx_carried_out()) {
        if (!recancentrate_controller_available()) {
            ASSERT(false);
        }
        /* else do nothing, wait for transmission to be carried
```

```
            * out */
    }

    return 0;
}
```

H.2. 0byte_transmitter.c

```
/*
 * 0byte_transmitter.c
 *
 * Transmits a 0 byte data frame, which is the shortest possible frame.
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "../recancentrate.h"
#include "../device_config.h"
#include "../assert.h"
#include "../common.h"
#include "../led.h"
#include "crash_controller.h"




int main(void)
{
    unsigned int sid = 1;
    t_recancentrate_status tx_status;

    enable_button_interrupts();
    init_leds();
    init_recancentrate_driver();

    led_display(tx_data[0]);
    request_recancentrate_tx(sid, NULL, 0, &tx_status);

    if (tx_status != RECANCENTRATE_STATUS_TX_REQUEST_SUCCESSFUL) {
        ASSERT(false);
    }

    while (!recancentrate_tx_carried_out()) {
        if (!recancentrate_controller_available()) {
            ASSERT(false);
        }
        /* else do nothing, wait for transmission to be carried
         * out */
    }

    return 0;
}
```

I. Source code for the profiler

I.1. profiler.c

```
/*
 * profiler.c
 *
 * Provides timers that can be started and stopped between two points in the
 * source code in order to measure how many instructions cycles have elapsed
 * between the two points. Inspect the contents of register TMR2, TM3, TMR4, or
 * TMR5 to view the number of instruction cycles that have been measured.
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifdef PROFILE

#include <p30f6014A.h>
#include "interrupts.h"

/* The timers, in 16-bit mode, can count up to a maximum value of 0xFFFF.
 * Because the profiling timers are configured to increment by 1 with each
 * instruction cycle, the maximum number of instructions we can count with a
 * 16-bit timer coincides with the maximum value a 16-bit timer can count up to.
 */
static const unsigned int INSTRUCTION_COUNT_MAX = 0xFFFF;



#define __CONCAT(a, b) a ## b
#define __CONCAT3(a, b, c) a ## b ## c

#define DEFINE_SETUP_TIMER(timer_idx) \
\
static void __CONCAT(setup_timer, timer_idx)(void) \
{ \
    /* Configure timer to increment by one with each instruction */ \
    /* ... timer clock source is the instruction cycle clock, \
     * i.e. FCY = FOSC/4 */ \
    __CONCAT3(T, timer_idx, CONbits).TCS = 0; \
    /* ... do not prescale the timer clock source */ \
    __CONCAT3(T, timer_idx, CONbits).TCKPS = 0; \
\
    /* Do not use the gated time accumulation mode */ \
    __CONCAT3(T, timer_idx, CONbits).TGATE = 0; \
    /* In idle mode, stop timer */ \
    __CONCAT3(T, timer_idx, CONbits).TSIDL = 1; \
```

```
    /* Assign maximum timeout to timer period registers */ \
    __CONCAT(PR, timer_idx) = INSTRUCTION_COUNT_MAX; \
} \


DEFINE_SETUP_TIMER(2);
DEFINE_SETUP_TIMER(3);
DEFINE_SETUP_TIMER(4);
DEFINE_SETUP_TIMER(5);


void profiler_setup_timers(void)
{
    setup_timer2();
    setup_timer3();
    setup_timer4();
    setup_timer5();
}


#define DEFINE_PROFILER_START_TIMER(timer_idx) \
\
void __CONCAT(profiler_start_timer, timer_idx)(void) \
{ \
    /* Reset timer to zero */ \
    __CONCAT(TMR, timer_idx) = 0; \
    /* Set timer on */ \
    __CONCAT3(T, timer_idx, CONbits).TON = 1; \
} \


DEFINE_PROFILER_START_TIMER(2);
DEFINE_PROFILER_START_TIMER(3);
DEFINE_PROFILER_START_TIMER(4);
DEFINE_PROFILER_START_TIMER(5);


#define DEFINE_PROFILER_STOP_TIMER(timer_idx) \
\
void __CONCAT(profiler_stop_timer, timer_idx)(void) \
{ \
      __CONCAT3(T, timer_idx, CONbits).TON = 0; \
} \


DEFINE_PROFILER_STOP_TIMER(2);
DEFINE_PROFILER_STOP_TIMER(3);
DEFINE_PROFILER_STOP_TIMER(4);
DEFINE_PROFILER_STOP_TIMER(5);
```

```
#endif /* PROFILE */
```

I.2. profiler.h

```
/*
 * profiler.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _PROFILER_H_
#define _PROFILER_H_



void profiler_setup_timers(void);

void profiler_start_timer2(void);
void profiler_stop_timer2(void);

void profiler_start_timer3(void);
void profiler_stop_timer3(void);

void profiler_start_timer4(void);
void profiler_stop_timer4(void);

void profiler_start_timer5(void);
void profiler_stop_timer5(void);



#endif /* _PROFILER_H_ */
```

J. Stimulus files for the MPLAB SIM simulator

J.1. c1omission_c2rxb1.sbs

```
## SCL Builder Setup File: Do not edit!!

## VERSION: 3.60.00.00
## FORMAT:  v2.00.01
## DEVICE:  dsPIC30F6014A

## PINREGACTIONS
cyc
No Repeat
IFS1.C1IF
C1INTF.TXB0IF
IFS2.C2IF
C2INTF.RXB1IF
C2RX1DLC.DLC
—
4348


1
1
1000
—
&
## ADVPINREGACTIONS
—
&
—
COND1
Any




—
&
## CLOCK
&
## STIMULUSFILE
```

```
&
## RESPONSEFILE
&
## ASYNCH
&
## ADVANCEDSCL

1
&
```

J.2. c1rxb0_c2rxb0_c1ewarn.sbs

```
## SCL Builder Setup File: Do not edit!!

## VERSION: 3.60.00.00
## FORMAT: v2.00.01
## DEVICE: dsPIC30F6014A

## PINREGACTIONS
cyc
No Repeat
IFS1.C1IF
C1INTF.RXB0IF
C1RX0DLC.DLC
IFS2.C2IF
C2INTF.RXB1IF
C2RX1DLC.DLC
C1INTF.EWARN
C1INTF.ERRIF
__
3302
1
1
1000




__
3302



1
1
1000


__
3998
1
```

```
1
1
--
&
## ADVPINREGACTIONS
--
&
--
COND1
Any




--
&
## CLOCK
&
## STIMULUSFILE
&
## RESPONSEFILE
&
## ASYNCH
&
## ADVANCEDSCL

1
&
```

J.3. c1rxb0_c2rxb1.sbs

```
## SCL Builder Setup File: Do not edit!!

## VERSION: 3.60.00.00
## FORMAT:  v2.00.01
## DEVICE:  dsPIC30F6014A

## PINREGACTIONS
cyc
No Repeat
IFS1.C1IF
C1INTF.RXB0IF
C1RX0DLC.DLC
IFS2.C2IF
C2INTF.RXB1IF
C2RX1DLC.DLC
--
```

```
4340
1
1
1000



__
4344



1
1
1000
__
&
## ADVPINREGACTIONS
__
&
__
COND1
Any




__
&
## CLOCK
&
## STIMULUSFILE
&
## RESPONSEFILE
&
## ASYNCH
&
## ADVANCEDSCL

1
&
```

J.4. c1txb0_c2omission.sbs

```
## SCL Builder Setup File: Do not edit!!

## VERSION: 3.60.00.00
## FORMAT: v2.00.01
## DEVICE: dsPIC30F6014A

## PINREGACTIONS
```

```
cyc
No Repeat
IFS1.C1IF
C1INTF.TXB0IF
--
4348
1
1
--
&
## ADVPINREGACTIONS
--
&
--
COND1
Any




--
&
## CLOCK
&
## STIMULUSFILE
&
## RESPONSEFILE
&
## ASYNCH
&
## ADVANCEDSCL

1
&
```

J.5. c1txb0_c2rxb1.sbs

```
## SCL Builder Setup File: Do not edit!!

## VERSION: 3.60.00.00
## FORMAT:  v2.00.01
## DEVICE:  dsPIC30F6014A

## PINREGACTIONS
cyc
No Repeat
IFS1.C1IF
C1INTF.TXB0IF
IFS2.C2IF
C2INTF.RXB1IF
C2RX1DLC.DLC
```

```
__
4340
1
1



__
4348


1
1
1000
__
&
##  ADVPINREGACTIONS
__
&
__
COND1
Any




__
&
##  CLOCK
&
##  STIMULUSFILE
&
##  RESPONSEFILE
&
##  ASYNCH
&
##  ADVANCEDSCL

1
&
```

Bibliography

Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. ISSN 1545-5971. doi: http://doi.ieeecomputersociety.org/10.1109/TDSC.2004.2.

Manuel Barranco. Improving error containment of controller area network (CAN) by means of adequate star topologies. Technical report, Universitat de les illes balears, Nov. 2008.

Manuel Barranco. *Improving Error Containment and Reliability of Communication Subsystems Based on Controller Area Network (CAN) by Means of Adequate Star Topologies*. PhD thesis, 2010.

Manuel Barranco, Julián Proenza, Guillermo Rodríguez-Navas, and Luís Almeida. CANcentrate: An active star topology for can networks. In *IEEE International Workshop on Factory Communication Systems*, pages 219–228, Sept. 2004.

Manuel Barranco, Luís Almeida, and Julián Proenza. ReCANcentrate: a replicated star topology for CAN networks. In *10th IEEE International Conference on Emerging Technologies and Factory Automation, 2005. ETFA 2005.*, volume 2, Catania, Italy, Sept. 2005a. doi: 10.1109/ETFA.2005.1612714.

Manuel Barranco, Julián Proenza, Guillermo Rodríguez-Navas, and Luís Almeida. A CAN hub with improved error detection and isolation. In *10th International CAN Conference (ICC 2005)*, Rome, Italy, March 2005b.

Manuel Barranco, Julián Proenza, and Luís Almeida. Experimental assessment of ReCANcentrate, a replicated star topology for CAN. *SAE 2006 Transactions Journal of Passenger Cars: Electronic and Electrical Systems*, 2006a.

Manuel Barranco, Julián Proenza, Guillermo Rodríguez-Navas, and Luís Almeida. An active star topology for improving fault confinement in CAN networks. *IEEE transactions on industrial informatics*, 2(2):78–85, May 2006b.

Manuel Barranco, Julián Proenza, and Luís Almeida. Management of media replication in ReCANcentrate. Technical report, Oct. 2007.

Manuel Barranco, Julián Proenza, and Luís Almeida. Designing and verifying media management in ReCANcentrate. In *Proceedings of the 13rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008), Hamburg, Germany*, Sept. 2008.

Manuel Barranco, David Geßner, Julián Proenza, and Luiís Almeida. First prototype and experimental assessment of media management in recancentrate. In *ETFA 2010. 15^{th} IEEE International Conference on Emerging Technologies and Factory Automation, Bilbao, Spain*, 2010.

Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. November 2004. URL www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

Robert Bosch GmbH. CAN specification version 2.0. Technical report, Robert Bosch GmbH, 1991. URL http://www.semiconductors.bosch.de/pdf/can2spec.pdf.

John Catsoulis. *Designing Embedded Hardware, Second Edition*. O'Reilly Media, 2005. URL http://www.oreilly.com/catalog/dbhardware.

Jay Fenlason and Richard Stallman. The gnu profiler. URL http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html.

Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

Barry W. Johnson. *Design and analysis of fault tolerant systems*. Addison-Wesley Publishing Company, Inc., 1989.

Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/357172.357176. URL http://research.microsoft.com/users/lamport/pubs/byz.pdf.

Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

Microchip. PIC18FXX8 Data Sheet - 28/40-Pin High-Performance, Enhanced Flash Microcontrollers with CAN Module, 2004.

Microchip. *MPLAB C30 C Compiler User's Guide*, 2005a.

Microchip. *MPLAB ASM30, MPLAB LINK30 and utilities user's guide*, 2005b.

Microchip. *dsPIC30F6011A/6012A/6013A/6014A Data Sheet*, 2006a.

Microchip. *dsPIC30F Family Reference Manual*, 2006b.

Microchip. *dsPICDEM 80-Pin Starter Development Board User's Guide*, 2006c.

Microchip. *MPLAB IDE User's Guide with MPLAB Editor and MPLAB SIM Simulator*. Microchip, 2009.

Philips Semiconductors. *APPLICATION NOTE PCA82C250 / 251 CAN Transceiver*. Philips Semiconductors, October 1996.

Philips Semiconductors. *PCA82C250 CAN controller interface*, 1997.

Juan Pimentel, Julián Proenza, Luis Almeida, Guillermo Rodíguez-Navas, Manuel Barranco, and Joachim Ferreira. *Automotive Embedded Systems Handbook (Industrial Information Technology)*, chapter Chapter 6: Dependable Automotive CANs. CRC Press, 2008.

Julián Proenza. *RCMBnet: A Distributed Hardware and Firmware Support for Software Fault Tolerance*. PhD thesis, Universitat de les illes balears, 2007.

Julián Proenza. Convocatoria de ayudas de proyectos de investigación fundamental no orientada. Technical report, 2009.

Julián Proenza and José Miro-Julia. MajorCAN: A modification to the controller area network protocol to achieve atomic broadcast. *IEEE International Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.

J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 150, Washington, DC, USA, 1998. IEEE Computer Society.

J. Rufino, P. Veríssimo, and G. Arroz. A Columbus' egg idea for CAN media redundancy. In *Digest of Papers, The 29th International Symposium on Fault-Tolerant Computing Systems*. IEEE, June 1999.

Julian Seward, Nicholas Nethercote, Jeremy Fitzharding, Tom Hughes, Josef Weidendorfer, Paul Mackerras, Greg Parker, Dirk Mueller, Robert Walsh, Bart Van Assche, Cerion Armour-Brown, Donna Robinson, Vince Weaver, Frederic Gobry, Daniel Berlin, Michael Matz, Simon Hausmann, and David Woodhouse. Valgrind. URL http://valgrind.org.

Wilfried Voss. *A Comprehensible Guide to Controller Area Network*. Copperhill Technologies Corporation, 2005.

Wikipedia. Prescaler — wikipedia, the free encyclopedia, 2009. URL http://en.wikipedia.org/w/index.php?title=Prescaler&oldid=319040906. [Online; accessed 2-June-2010].

Wikipedia. Field-programmable gate array — wikipedia, the free encyclopedia, 2010a. URL http://en.wikipedia.org/w/index.php?title=Field-programmable_gate_array&oldid=339015623. [Online; accessed 1-February-2010].

Wikipedia. Profiling (computer programming) — wikipedia, the free encyclopedia, 2010b. URL http://en.wikipedia.org/w/index.php?title=Profiling_(computer_programming)&oldid=364322500. [Online; accessed 25-June-2010].

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

XESS Corporation. *XSA-3S1000 Board V1.1 User Manual*, Sept. 2007.

XESS Corporation. What are CPLDs and FPGAs?, 2010. URL `http://www.xess.com/appnotes/fpga_tut.php`. [Online; accessed 1-February-2010].

Holger Zeltwanger. Failure detection and error handling in CAN-based networks. In *Seminario Anual de Automática, Electrónica Industrial e Instrumentación*, Pamplona, Spain, September 1998.

Index