Universitat de les Illes Balears

escola politècnica superior

Treball Final de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Implementation and Verification of the Slave Elementary Cycle Synchronization Mechanism of the Flexible Time-Triggered Replicated Star for Ethernet

INÉS ÁLVAREZ VADILLO

Tutors

Julián Proenza Manuel Barranco

Supervisor

Alberto Ballesteros

Escola Politècnica Superior

Universitat de les Illes Balears

Palma, September 24, 2014

CONTENTS

LIST OF FIGURES

List of Tables

ACRONYMS

AW *asynchronous window*

COTS *commercial off-the-shelf*

DES *distributed embedded system*

EC *elementary cycle*

FT4FTT *fault tolerance for flexible time-triggered*

FTT *flexible time-triggered*

FTT-SE *flexible time triggered-switched Ethernet*

FTTRS *flexible time-triggered replicated star*

QoS *quality of service*

SECSM *slave elementary cycle synchronization mechanism*

SW *synchronous window*

TAW *turn around window*

TM *trigger message*

TMW *trigger message window*

ABSTRACT

A distributed embedded system that must work in dynamic environments, where the working conditions may change in an unpredictable way, must be flexible. Moreover, if such system must operate continuously it must also be reliable. Therefore, its internal communication network must support both attributes. Unfortunately, nowadays it does not exist any communication network that completely fulfils both requirements.

The *Flexible Time-Triggered* (FTT) communication paradigm provides the system with support for time-triggered and event-triggered communication, as well as adaptability to environment changes. FTT can be deployed on top of any existing network technology. Implementing FTT on top of the Ethernet communication protocol offers a high bandwidth and a low cost of the components of the network. Nevertheless, FTT-Ethernet lacks the necessary fault-tolerance mechanisms to provide adequate dependability levels.

The *Fault Tolerance for Flexible Time-Triggered Ethernet* (FT4FTT) research project faces the development of the required fault tolerance mechanisms in order to construct a flexible and highly-dependable communication infrastructure based on FTT-Ethernet.

The *Flexible Time-Triggered Replicated Star* (FTTRS) was proposed as part of the FT4FTT project to provide adequate fault-tolerance mechanisms for the communication infrastructure. Among other features, FTTRS provides tolerance in front of transient faults in the channel. This is done by means of time and information redundancy. The *Slave Elementary Cycle Synchronization Mechanism* (SECSM) is a proposal to manage such redundancy that pays a special attention to achieve a proper synchronization among the nodes of the system. In this way all nodes have a similar view, even in the presence of faults, of when the different communication cycles start.

This project consisted in the implementation, validation and modification of the SECSM; as well as the later evaluation of the complete prototype that was built. The results of this project have been published in a conference paper that is attached as an appendix.

CHAPTER 1

INTRODUCTION

1.1 Background and motivation

An embedded system is a computer system designed with the aim of controlling a different system. The reduction in the cost of processors has facilitated the use of this type of systems in many different fields, so that it is possible to find them integrated in factories, buildings, vehicles and a wide variety of equipment.

Embedded systems are usually subjected to weight, size and energy constrains. Moreover, because of their nature embedded systems are real-time systems [1], that is, they have to react to the environment within a limited amount of time. This implies the usage of specific scheduling policies [2]. Also, when the application is critical embedded systems must provide high reliability. This can be achieved by adding fault tolerance mechanisms which need to be carefully evaluated.

Environment restrictions frequently require that the system is distributed among different nodes. When this happens the system is called a *distributed embedded system* (DES). DESs require the usage of a communication network for information exchange. Traditionally DESs have been used in static environments, which means that the operation conditions are known in advance and, thus, the use of offline scheduling mechanisms is an adequate choice.

However, nowadays it is pursued the introduction of DESs in dynamic environments where the operation conditions may change in an unpredictable way. In these cases using static scheduling approaches is not a suitable solution, as it can present two disadvantages [3]. On the one hand, it can lead to the waste of resources when assuming worst case conditions. On the other hand, if average conditions are assumed some components can become overloaded because of unpredicted situations, what can bring violations of the real-time or the dependability requirements.

This new tendency has led to the appearance of the concept of *adaptive embedded systems*. A system is said to be adaptive when it is capable of automatically adjusting its behaviour to respond to changes in the environment. Thus, adaptive embedded systems must be both flexible and reconfigurable, and require support for such attributes

at various levels of the architecture, including the operating system and, in the case of DES, the communication network. However, nowadays it does not exist any network technology that completely fulfils such requirements, as traditional networks support either time-triggered or event-triggered communication [1] [4], and those that support both paradigms, as is the case of FlexRay [5], do not provide mechanisms to manage dynamic changes in the communication requirements.

The *flexible time-triggered* (FTT) [4] paradigm is a proposal for unifying time-triggered and event-triggered communication into a single technology. Moreover, FTT provides support for changing the communication parameters to satisfy the changing traffic needs, while at the same time not violating the real-time requirements of the application. It is based on a mechanism that schedules the communication online, which supports changes in the traffic, and in a centralized manner, which facilitates the implementation of the scheduling policies. The node that implements the scheduling mechanism is called *master*, whereas the regular DES nodes –that is, the nodes that implement regular applications of the DES– are called *slaves*. The communication is divided in time slots of fixed duration called *elementary cycle* (EC). Each EC is divided in a synchronous window, in which synchronous messages are transmitted, and an asynchronous window, in which asynchronous messages are transmitted. In each EC the master decides which slaves will transmit according to the scheduling policies and the particular conditions. The master polls the slaves by broadcasting the so called *trigger message* (TM), that is also used by the slaves to determine the start of the following EC.

In principle, FTT can be deployed on top of any network technology [4]. Implementing FTT on top of Ethernet seems a suitable approach due to the low cost of the components and the high bandwidth offered by said communication protocol. This has been done in the form of the FTT-Ethernet protocol [6], one of whose fundamental variants is *flexible time triggered-switched Ethernet* (FTT-SE) [7]. Unfortunately, FTT-Ethernet still lacks the necessary fault tolerance mechanisms to guarantee that the adequate dependability levels are achieved.

The *fault tolerance for flexible time-triggered* (FT4FTT) project [8] tackles the construction of a highly-dependable and flexible communication infrastructure based on the FTT-Ethernet protocol. Specifically, FT4FTT faces the development of the required fault tolerance mechanisms and their later integration in order to build a complete system architecture.

In particular, one of the problems of FTT-Ethernet is the lack of adequate mechanisms for facing faults in the channel that affect the transmission of specific messages. For this reason one of the tasks entailed in FT4FTT consists in introducing time and information redundancy in order to tolerate transient faults in the communication channel.

The *flexible time-triggered replicated star* (FTTRS) [9] was proposed as part of the FT4FTT project to, among other things, tolerate said transient faults. The complete architecture of FTTRS is not relevant for the development of this project. However, what is important is that FTTRS includes mechanisms to provide time redundancy. Specifically, FTTRS proposes that the master transmits the TM several times at the beginning of each EC. Each of these TMs are called a *replica*. If the number of replicas is sufficiently high, we can guarantee that at least one replica will reach the slaves, allowing for synchronizing them even in presence of transient faults in the communication channel.

Nevertheless, transient faults do not necessarily affect to all the slaves in the same way, which can cause time inconsistency in the reception of the TM replicas. That is, each slave can receive a different replica of the TM. Since each TM replica is transmitted in a different moment, slaves that receive different replicas will determine the start of the following EC at different moments too. The *slave elementary cycle synchronization mechanism* (SECSM) [10], which is part of the FTTRS architecture, was designed to solve this problem.

Taking this as point of departure, the purpose of this project is to implement the SECSM so as to obtain a prototype in order to make an experimental evaluation of the design and the implementation. Specifically, we want to obtain experimental results in order to measure the accuracy in the synchronization –meaning the deviation of the EC start moment in each slave– achieved by the implementation when transient faults in the channel do occur.

1.2 Goals of the project

The background and motivation previously described provides us with the bases to understand the goals of this project.

This Bachelor Thesis has two specific goals:

- Implementing the slave elementary cycle synchronization mechanism in order to build a functional prototype.
- Measuring the accuracy of the synchronization using fault injection in order to evaluate the design and the implementation of the slave elementary cycle synchronization mechanism.

1.3 Work that has been carried out

Even though in Chapter 3 we talk about the different phases and activities that conform this project, in this section we want to give an overview of the work realized during this project.

As previously said in Section 1.1, the SECSM [10] was proposed as part of the FT4FTT project. That is, we already had the specification of the system. Moreover, the starting point of the implementation was an already existing software implementation of the FTT-SE protocol [7] [11]. That is, we were provided with a development platform to which we integrated the SECSM.

Protocol study

In order to carry out this project we had to study the FTT communication paradigm, the FTT-SE communication protocol and the SECSM mechanism. Also, it was necessary to become familiar with the development platform in order to identify the files and functions that would be modified during the implementation.

It is noteworthy that this study was carried out during the first semester of the course 2013–2014.

Implement the SECSM in the Master

As it is later explained in Section 2.3, in FTTRS the TM is sent by the master several times in each elementary cycle. The slaves use the set of TM replicas that they receive to synchronize. In order to allow for the slaves to synchronize it was necessary to modify the TM header to convey some information. Moreover, the development platform was also modified to implement the transmission of multiple TM in each EC.

Implement the SECSM in the Slave

As it is later explained in Section 2.3, we have to program each slave to manage the redundancy in the reception of the TMs, as well as synchronize the start moment of the following EC among the slaves from the set of TM replicas received.

Propose the inclusion of the Turn Around Window in the specification

In a first specification of the SECSM, the time needed by the slaves to process the TM replicas was not taken into account. In order to allow the slaves to decode and process the TM replicas before the transmission of synchronous messages starts, we proposed the adaptation of the SECSM specification so as to add the Turn Around Window. As later explained in Section 2.2, the *turn around window* (TAW) is a specific amount of time during which slaves process the last TM replica in case they receive it.

Evaluate the SECSM implementation

We also carried out an exhaustive verification of the SECSM implementation in order to assess its correctness and measure the accuracy reached in the synchronization. In order to do so we used software implemented fault injection, which required adding some code to the slave software to inject transient faults in the channel.

Write a short paper

The results obtained in this project permitted the writing of a short paper, in which the candidate participated as co-author together with David Gessner, Alberto Ballesteros, Manuel Barranco and Julián Proenza. The reference of the mentioned paper is:

- Gessner, D., Álvarez, I., Ballesteros, A., Barranco, M., Proenza, J., *Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet.* In Proc. 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), September 16-19, 2014, Barcelona, Spain.

1.4 Document structure

The reminder of this document is divided in seven chapters. Chapter 2 briefly describes the previous work that conforms the bases for this project, giving the necessary foundations for the rest of the work. This chapter includes an overview of the FTT-SE protocol on which FTTRS is based, as well as the design of the SECSM. Chapter 3 discusses the

phases in which this project is divided and the organization of the work. Chapter 4 describes the results obtained from studying the FTT-SE prototype we used as starting point of this project.

Chapter 5 and 6 report on the implementation and test of the SECSM in the master and the slave respectively. Chapter 7 describes the results obtained from testing the integration and the exhaustive verification of the whole implementation.

Chapter 8 contains a summary of the most relevant aspects of this project, as well as some suggestions of future work and some considerations about the learning process of the bachelor candidate.

Finally Appendix A contains the code of the implementation and Appendix B contains the the published short paper, which was previously mentioned in Section 1.3.

CHAPTER 2

PREVIOUS WORK

This chapter briefly explains the previous work that constitutes the starting point of this project. It describes the basics of the Flexible Time-Triggered paradigm (FTT), the FTT Switched Ethernet protocol (FTT-SE) and explains the design of the slave EC synchronization mechanism.

2.1 The Flexible Time-Triggered paradigm (FTT)

The Flexible Time-Triggered paradigm (FTT) [4] is a communication paradigm proposed at the University of Aveiro (Portugal) to support event and time-triggered communication in a flexible manner. That is, FTT provides mechanisms for dynamically change the communication requirements.

FTT follows a master/multi-slave scheme. As already explained in Section 1.1, the communication is managed and coordinated by a special node called master. The rest of the nodes of the DES are the slaves and they are regular nodes –they execute the control application and communicate according to the instructions of the master–.

The master organises the communication in periods of constant duration called elementary cycles (EC). The structure of the EC can be seen in Figure 2.1. The master polls several slaves by means of a single message, called trigger message (TM), which conveys the scheduling information of each EC. Note that the reception of the TM at the beginning of the EC allows synchronizing the communication among the slaves.

The rest of the EC is divided into two different phases, the *synchronous window* (SW), where the time-triggered communication is carried out; and the *asynchronous window* (AW), where the event-triggered communication takes place. From now on, we will refer to the time-triggered communication as synchronous, as the transmission period each message is synchronized with the EC duration, and to the event-triggered communication as asynchronous, as it happens asynchronously with the EC.

It is noteworthy that since the master schedules the communication at the beginning of every EC, the communication requirements can not be changed within a given

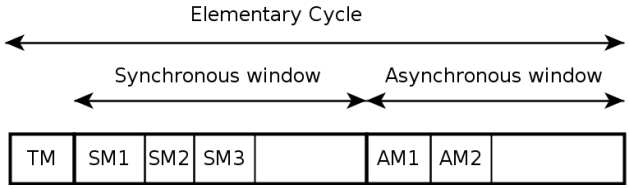


Figure 2.1: EC structure. Synchronous and asynchronous messages are labelled as SMi and AMi respectively.

EC. Conversely, slaves can ask for changes in the communication requirements during the asynchronous window of each EC. The master accepts or rejects the changes depending on the availability of the resources. If the changes are accepted, the master informs about the resources assigned to each kind of traffic during the asynchronous window of the following EC. After notifying the changes to the slaves, the master also take them into account when doing the schedule from then on.

Finally, FTT can be deployed on top of different existing communication protocols. So far, the FTT paradigm has been implemented on top of CAN [4] and Ethernet. In particular, there are different FTT Ethernet realizations [12, 13, 14]. Both FT4FTT and this bachelor thesis build upon FTT-Switched Ethernet, as it the topology that is currently receiving more attention from the research community.

2.2 FTT-Switched Ethernet (FTT-SE)

FTT-Switched Ethernet (FTT-SE) is a realization of the FTT paradigm on top of Switched Ethernet that presents several important advantages. Two of the main advantages of Switched Ethernet are its high bandwidth and its extended application. This last advantage is crucial, as it makes it easier to access to low cost components and facilitates the integration of the FTT mechanisms within the rest of the system.

The FTT-SE architecture includes a commercial Ethernet full-duplex switch. Each node is connected to the switch by means of two different links: the uplink, used for transmitting, and the downlink, used for receiving. Since the communication is full-duplex, each slave can receive and transmit information at the same time, and the bandwidth utilization can be improved by transmitting in parallel through disjoint communication paths. Moreover, as the switch provides a collision-free communication domain, nodes no longer need to implement collision avoidance functionalities.

In FTT-SE the communication is still divided in ECs and the synchronous and asynchronous traffic are separated in two different windows. Even though, the EC structure has suffered various modifications with respect to the originally described in FTT.

As depicted in Figure 2.2, each EC starts with the *guard window*, which is the time used by the master to transmit the TM. Its duration corresponds to the maximum length of the TM. The TM size varies depending on the synchronous traffic transmitted each EC, which determines the size of the schedule.

Furthermore, FTT-SE includes the (TAW). This window is the time used by the slaves to process the TM. Its duration corresponds to the time needed by the slave with

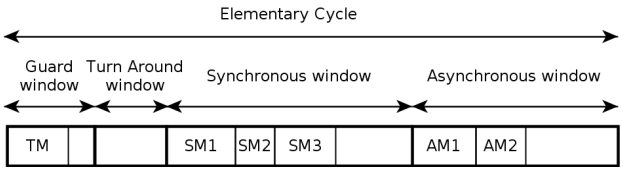


Figure 2.2: FTT-SE EC structure.

less computational capacity to decode the TM. This means, it is the maximum amount of time needed to decode the TM.

From the point of view of the nodes, FTT-SE is implemented as a middleware that is deployed between the application and the Ethernet layers, giving a higher level of abstraction as it offers a common application interface while the FTT features and operations are hidden.

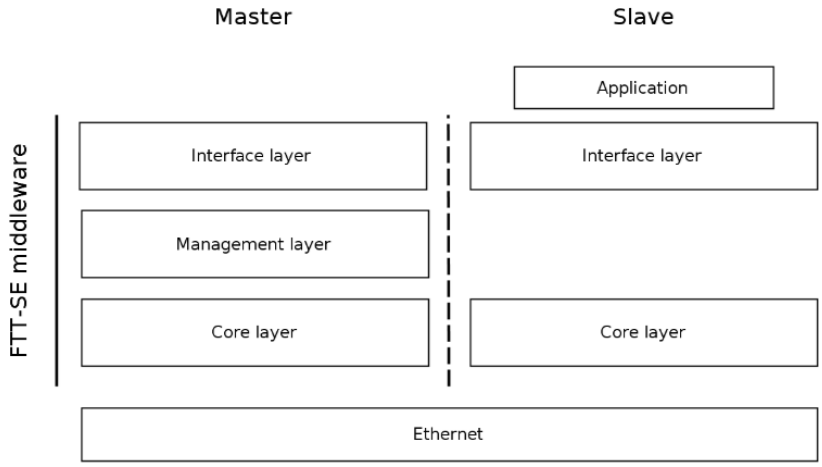


Figure 2.3: FTT-SE middleware architecture.

As shown in Figure 2.3, the FTT-SE middleware is divided into three layers: the *interface*, the *management* and the *core* layers. Next we briefly outline the features and functionalities of each of them.

- Interface layer: this layer provides the application software running on the nodes with common communication and management services. It is used by the slaves to setup and modify the communication and *quality of service* (QoS) parameters [12].

- Management layer: this layer is only present in the master. It manages the the resources devoted to each type of communication according to their QoS requirements.

- Core layer: this layer implements the communication mechanisms of the FTT protocol, i.e. in the master the core layer is the responsible of build up the schedule conveyed in the TM and carries out the transmission control.

2.3 The Slave Elementary Cycle Synchronization Mechanism (SECSM) specification

The SECSM is a part of the FTTRS [9] architecture. FTTRS aims at constructing a highly-dependable and flexible communication infrastructure based on FTT-SE by means of fault-tolerance mechanisms.

Specifically, FTTRS introduces, among other features, time and information redundancy in order to face transient faults in the communication channel. Specifically, the master transmits the TM several times at the beginning of each EC. Therefore, each EC starts with a *trigger message window* (TMW), where multiple TM replicas are sent. During this window no more information can be transmitted.

One of the main functions of the TM is to synchronize the start of the EC among all the slaves. The SECSM allows the slaves to synchronize using the new scheme. In FTTRS slaves are synchronized when they all consistently predict the TMW expiration time.

Each TM is transmitted k times in each EC, where k is a number that depends on the channel bit error rate. From the k replicas transmitted by the master in each EC, a slave can receive all of them or just a subset if transient errors do occur in the channel. The SECSM can synchronize the slaves regardless of which replicas reach each slave if the following requirements are accomplished: (a) each slave receives at least one TM replica per TMW; (b) the TM replicas transmission is done such that every slave that receives the same replica do so at the same time by each of their links; (c) the TM replicas are broadcast with the same constant inter-transmission time τ; (d) the amount of time by which the clock of non-faulty slaves can drift apart within one EC is negligible.

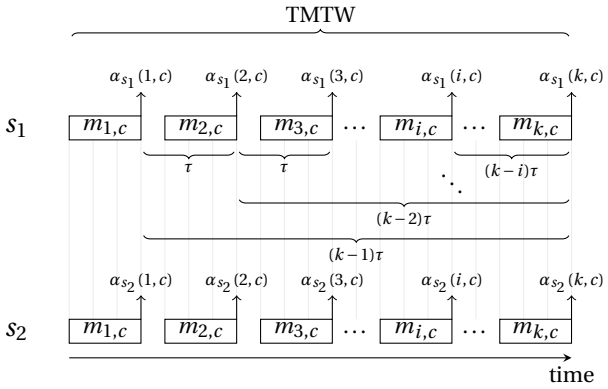


Figure 2.4: Aligment of TM arrival times.

Let c denote the sequence number of a certain EC and let S_c be the set of slaves that remain non-faulty at the end of EC c. Also, let $m_{i,c}$ denote the TM with sequence number i belonging to EC c, where $i \in \mathbb{N}$ and $1 \leq i \leq k$. Moreover, let $M_{s,c}$ be the TM replicas received by slave s during EC c. Finally, let $\alpha(i,c)$ denote the expected arrival time of $m_{i,c} \in M_{s,c}$ at slave $s \in S_c$.

Figure 2.4 shows the expected arrival times at salves $s_1, s_2 \in S_c$ of the k replicas sent in EC c.

The expected arrival time of $m_{k,c}$ at slave s is:

$$\alpha_s(k,c) = \alpha_s(i,c) + (k-i)\tau, \tag{2.1}$$

which is the end of the TMW, as is depicted by Figure 2.4.

$\alpha_s(k,c)$ can be calculated for all $s \in S_c$ for each c if at least one TM replica $m_{i,c}$ is received by s in c. Moreover, because of requirement (b) we also have that:

$$\alpha_{s1}(i,c) = \alpha_{s2}(i,c). \tag{2.2}$$

The SECSM can synchronize the TMW expiration time among s_1 and s_2 using as synchronization event the expected moment of the k TM replica reception.

CHAPTER 3

PHASES OF THE PROJECT

This project was organized into four phases, namely (1) the *specification* of the project, (2) the *study of the foundations and the available FTT-SE prototype*, (3) the *implementation and validation* and (4) the *documentation* (see Figure 3.1).

The *specification* of the project was carried out by the tutors and supervisors before the realization of the project itself. This phase comprises the design of the slave EC synchronization mechanism (SECSM), the identification of the different elements and modules of the FTTRS architecture related to the SECSM, and the proposal of the activities that are necessary to implement and validate the SECSM.

The phase devoted to the *study of the foundations and the available FTT-SE prototype* encompasses the activities carried out by the candidate to become familiar with the FTT-SE protocol and its available implementation, as well as with the SECSM. More specifically, some of the most relevant activities of this phase are the following ones:

- Study the FTT paradigm.
- Thoroughly comprehend the implementation of the *available* FTT-SE implementation. At this point it is important to note again that the implementation and validation of the SECSM has been carried out taking as a starting point an already available software implementation (middleware) [7] [11] of the FTT-SE protocol. Thus, it was necessary to study this middleware in detail to identify the different functions that will be involved in the implementation of the SECSM (see Section 4).
- Study the design of the SECSM.

As concerns the activities of the *implementation and validation* phase, they are the ones the candidate carried out to implement and validate the SECSM. These activities are outlined next.

- Implement and validate the multiple TM transmission mechanism in the master (see Section 1.3).

- Implement and validate the redundancy management mechanism in the slaves (see Section 1.3).
- Implement and validate the Turn Around Window. As it will be explained later, this activity was not foreseen in the specification of the project. It was a contribution of the candidate to propose this activity in order to tackle some issues that were not previously identified by the tutors and supervisors (see Section 6.3).
- Integrate the just mentioned implementations to obtain a complete SECSM prototype and, then validate it by means of fault injection.

It is noteworthy that these implementation and validation activities are organized following an incremental and iterative strategy. As depicted in Figure 3.1, each one of the three first activities of the *Implementation and validation* phase is divided into three steps. These steps are iteratively executed until the activity is considered to be fulfilled. The steps are the following ones:

- Specification: in this step the candidate studies the design of the mechanism addressed in the activity and, then, proposes a viable solution to implement it.
- Implementation: in this step the candidate implements the solution she proposed in the specification step.
- Testing: in this step the candidate describes, performs and analyzes a series of tests in order to assess the correctness of the implemented solution . Depending on the results of this assessment, it is decided whether or not the solution is valid, i.e. if it fulfils the requirements specified in the design of the mechanism. If not, the activity is resumed from the specification step to refine the implementation.

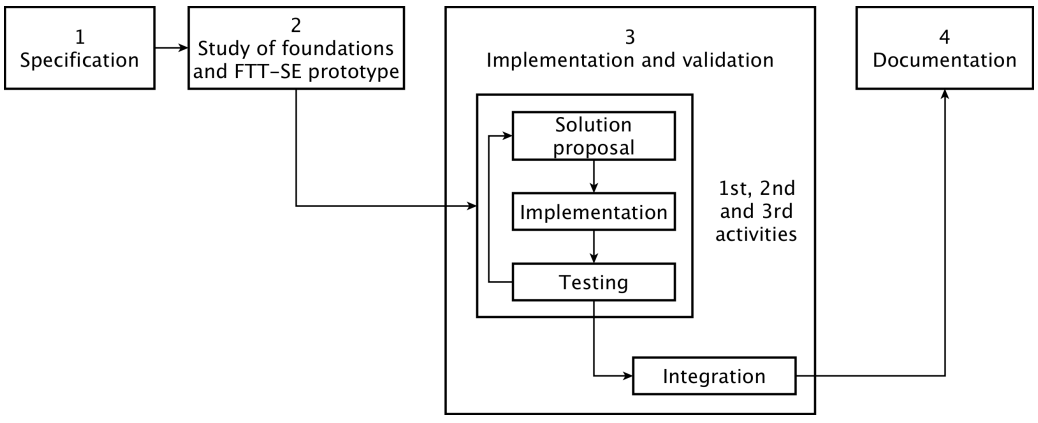


Figure 3.1: Phases, activities and steps of the project.

When each one of the three first activities of the *implementation and validation* is fulfilled, it is executed the fourth activity of this phase. As pointed out just before, during this activity the candidate integrates and tests as a whole the mechanisms she implemented in the other activities. In this way she obtains and validates the correctness of the complete prototype of the SECSM.

The last phase of the project, i.e. the *documentation*, includes two activities, namely the writing of the report that describes the project itself, and the submission (to an international conference) of a paper that summarizes the work and the results of the project.

The following Chapters and Sections describe in more detail the activities carried out in the second and third phases, i.e. *study of the foundations and the available FTT-SE prototype* and *implementation and validation*.

CHAPTER 4

STUDY OF THE AVAILABLE FTT-SE PROTOTYPE

The implementation of the SECSM has been carried out taking as starting point an available software implementation (middleware) of the previously described FTT-SE communication protocol (see Section 2.2). Thus, as said before, this available FTT-SE middleware constitutes the development platform of this project.

In order to add the SECSM to the middleware it was necessary not only to study in detail the FTT paradigm, but also to analyze and identify the different parts and functions of the middleware itself that are affected by the SECSM implementation.

The result of this analysis is shown in Figure 4.1, where we can find the files (depicted as circles) of the middleware which are relevant for this project.

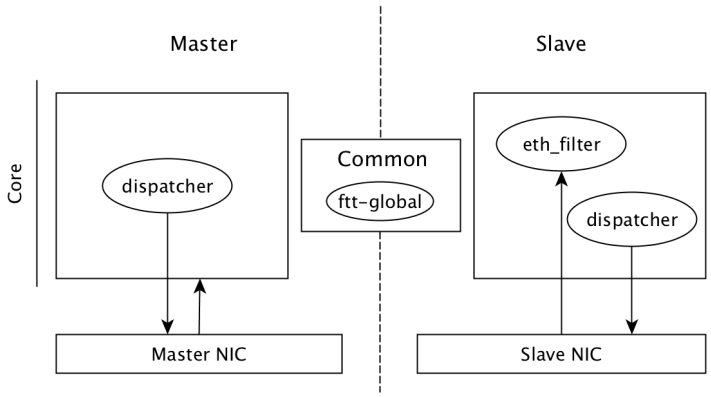


Figure 4.1: Files affected by the implementation of the SECSM.

At this point please refer to Figure 2.3, which shows the FTT-SE middleware architecture. As can be seen from this figure and Figure 4.1, the implementation of the

SECSM only affects specific files of the core layer of the FTT-SE middleware. Next we explain the functionality of each one of these files and the modifications that were necessary to carry out on them.

- Master `dispatcher`: this file contains the code responsible for the construction and transmission of the TMs. This means that it had to be adapted to send multiple TM replicas each EC with a given inter-transmission time τ.
- Slave `eth_filter`: when a message is received by a slave, this one is the file where the code for identifying the type of the message and extracting the information is placed. In this file we had to implement the management of the reception of several TM replicas in each EC as well as the synchronization mechanism according to what has been explained in Section 2.3 .
- Slave `dispatcher`: as it will be later explained in Section 6.3, some synchronization issues during the implementation of the mechanism leaded to the addition of the Turn Around Window –which has been previously explained in Section 2.2– in the specification of the mechanism.
- `ftt-global`: this file contains the specification of all the elements that are common to the master and the slave. This is the case of the TM header, which had to be modified to convey the information related to the mechanism.

Finally, the `ports` file, which is a library used by all the layers of the mechanism had to be modified too. We added to this file all the functions that are not part of the slave elementary cycle synchronization mechanism, but that are necessary to carry out the implementation.

CHAPTER 5

IMPLEMENTATION AND VALIDATION OF THE SECSM IN THE MASTER

Across this chapter we will explain the iterations carried out to complete all the activities related to the implementation and validation of the SECSM in the master.

5.1 First iteration: preliminary version of the multiple TM transmission in the master

As previously seen in Section 2.3, the master is in charge of sending k replicas of the TM each EC with a constant inter-transmission time τ. In order to add to the system the new functionalities offered by the SECSM, we had to encapsulate some additional information in the TM constructed by the master each EC. Moreover, the master code had to be modified in order to implement the transmission of the TM replicas.

In this section we find the solutions proposed for solving both problems and their implementation. Also we find the test setup used to verify the correctness of the implemented solutions and the results obtained from its performance.

5.1.1 Solution proposal and implementation

In this first solution, we modified the ftt-global.h file in order to add to the TM the corresponding information. Specifically, we added one byte to specify the number of TM replicas per EC k, two bytes for the TM inter-transmission time τ in μs and an additional byte that indicates the sequence number of each TM replica. Figure 5.1 shows the resulting TM header, where the highlighted fields correspond to the added information.

In order to ease the utilization of the prototype, the user must be able to modify k and τ without having to change the code. For this reason k and τ are previously specified by the user as parameters of the master process. Since the FTT-SE middleware

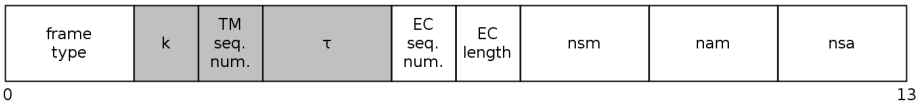


Figure 5.1: TM header including the information related to the SECSM.

is conformed by several layers, we had to modify all the functions that are built on the top of the master dispatcher.

Since the EC duration is also a parameter specified by the user, it is necessary to ensure that it will be possible to carry out the transmission of all of the TM replicas within an EC. That is, it has to be accomplished that:

$$k \times \tau < EC_{length}. \tag{5.1}$$

Moreover, as it is explained in Section 2.3, at least two TM replicas have to be sent in each EC. Both conditions are checked during the initialization of the master dispatcher and, in case any of them is not accomplished, the process finishes with an error message with the adequate information.

Regarding the transmission of the TM replicas, the code had to be adapted in order to make it possible to send every TM several times. Next we detail how the construction and the transmission are performed.

All the TM replicas carry the exact same information except for the TM sequence number, which is incremented every iteration and it can take values from 0 to $k-1$. Nevertheless, each time a message is sent, first, it is necessary to reserve the buffer where the message will be saved until its transmission. Once the message has been transmitted, the buffer is emptied and its content is lost. Therefore, it is necessary to construct the whole TM every time a replica has to be transmitted. For this reason, both the construction and the transmission of each TM replica are now carried out in a loop which is executed k times. Once the TM is constructed it has to be sent.

Let M_c denote the set of TM replicas belonging to the EC c. Let consider the transmission of the TM replica with sequence number i in EC c, where $i \in \mathbb{N}$ and $0 \leq i < k-1$, denoted by $m_{i,c}$, and the one with sequence number j, where $j \in \mathbb{N}$ and $j = i+1$, denoted by $m_{j,c}$, where $m_{i,c}, m_{j,c} \in M_c$. Let $tx(i,c)$ be the transmission moment of $m_{i,c}$ and $\beta(j,c)$ be the estimated transmission moment of $m_{j,c}$.

When the master process transmits $m_{i,c}$ it timestamps the moment of the transmission, $tx(i,c)$, and then τ is added to this moment in order to calculate $\beta(j,c)$:

$$\beta(j,c) = tx(i,c) + \tau. \tag{5.2}$$

Once the jth TM replica is constructed the process sleeps and it awakes when $\beta(j,c)$ arrives. Next we find the pseudo-code for this implementation.

```
function dispatcher is
    loop
        sleep_until_next_EC();
        tx(i,c) = now;
        c++;
        for i in k do
            m(i,c) = construct_TM();
            sleep_until(β(i,c));
            transmit(m(i,c));
            β(i+1,c) = tx(i,c) + τ;
        end for
    end loop
end function
```

5.1.2 Testing

In order to assess the correctness of the implemented solution it is necessary to ensure that each component involved in the mechanism operation works properly. For this purpose we defined several tests, with their respective setups, to evaluate each part and their integration.

All the setups have several features in common. On the one hand, all the tests are executed in the same machine and under the same GNU/Linux instance. Even the purpose of this mechanism is to work in an FTTRS [9] implementation with replicated masters, the master synchronization was not implemented when this project started. Therefore, and because of requirement (b) explained in Section 2.3, we considered that using a single master to test the performance was a suitable solution for testing the results of this implementation. Moreover, implementing the slaves in the same machine provides the slaves with a common clock base for timestamping. On the other hand, master and slaves processes are all attached to a 100 Mbps Ethernet switch.

The main differences between setups are the number of slaves involved and the network configuration. The number of slaves varies depending on the state of the implementation. Regarding the network construction, in the first setups we use Virtual Distributed Ethernet (VDE) [15]. VDE allows for building completely software based networks, distributed and Ethernet compliant. The prototypes used to carry out the tests use two VDE components, the switch, which is a process executed in the same machine as the master and slave processes, and virtual dedicated links that interconnect each node to the switch. Using VDE networks allows for seeing how the SECSM performs without taking into account the delays introduced by the network interfaces and the switch.

We added some instrumentation code to both the master and the slave software in order to be able to make measurements.

As concerns the test parameters, which are also shared by all the setups, we decided to use values that could be used by a control application. Typically a control application needs to transmit small amounts of information with small transmission periods corresponding to the sampling periods of the application. For this reason, the EC duration was set to 1 ms. Moreover, the inter-transmission time τ must be greater than the transmission time of a TM, including the Ethernet interframe gap. In these

tests the TMs do not carry any scheduling information, so as seen in Figure 5.1 the data payload of a TM is 14 bytes. Therefore, TMs fit within the 46 bytes of data padding of an Ethernet frame of minimum size. Figure 5.2 shows the format of the Ethernet MAC frame.

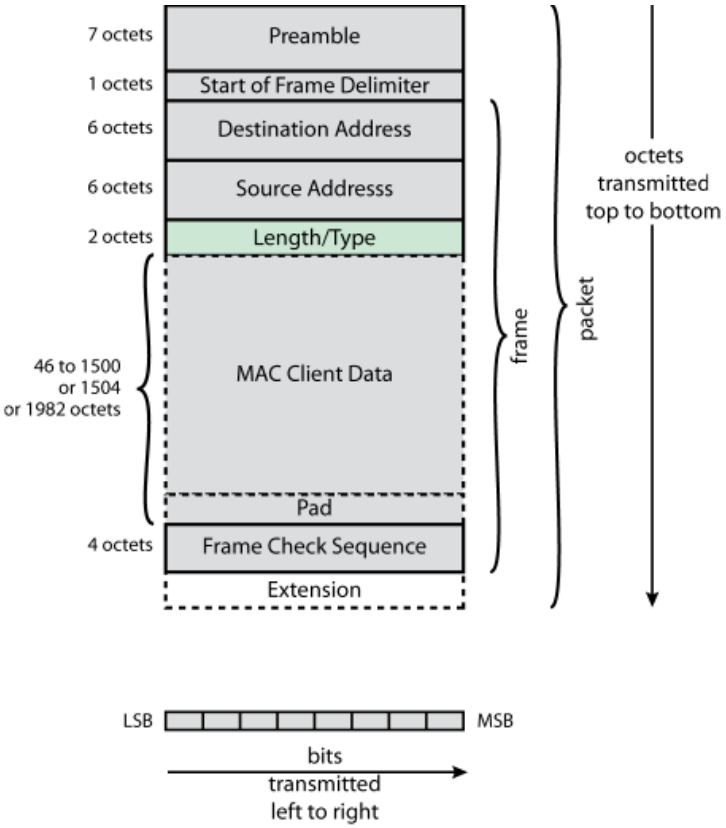


Figure 5.2: IEEE 802.3 MAC Frame Format [16].

This means that with 100 Mbps Ethernet the transmission time of a TM is $(72 \cdot 8)/100 = 6.72\mu$s. Therefore, τ must be greater than 6.72μs. Given that during the TM window only TMs can be exchanged, if τ is big enough the TMs will not be queued in the switch output ports, reducing the jitter introduced by the transmission. This way, not only the master will send the TM replicas with the adequate inter-transmission time, but also the slaves will receive them with the right τ. In order to reduce the impact of the non-determinism introduced by the OS and the software components and to allow slaves to process the replicas, we decided to set τ to 100 μs to perform the tests.

Regarding the number of TM replicas k is a function of the bit error rate in the channel which means k is implementation dependent. Assuming k is big enough to guarantee that each slave will receive at least one TM replica in each EC, it is not necessary to check the right functioning of the implementation for every value k can take. Specifically, to carry out our tests the value of k chosen is 4. The reason to choose this value is that a k of 4 allows us to study a significant number of scenarios without having a too elevate number of retransmissions.

Specifically, in order to assess the correct functioning of the solution we used to

implement the SECSM in the master we had to test two aspects.

First, we had to ensure that the information contained in the TM replicas was correct. In order to do so, we needed to check that the information is correctly encapsulated in the TMs. To this aim we used a network analyser, Wireshark, that allows for capturing the frames in the network.

The second part of the test consisted in measuring the time gone by between the transmission of the TM replicas with sequence number i where $0 \leq i < k-1$ and the one with sequence number j where $j = i+1$, both belonging to the EC c. In order to do so, we saved the transmission instant of the TM replicas i and j, denoted as $tx(i,c)$ and $tx(j,c)$ respectively. We obtained the real inter-transmission time by calculating the difference between the transmission moments.

Figure 5.3 shows the prototype used to carry out this test. This prototype is conformed by a master process attached to a VDE link and a Wireshark process, all executed in the same machine.

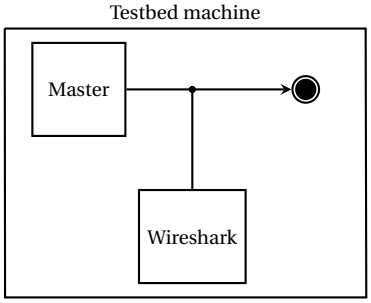


Figure 5.3: Prototype architecture with a master process and a network analyser.

As the master process is the only node in the configuration, having a switch was not necessary. Therefore, the master was attached to a virtual link that drives the frames to a sink.

The results of this test showed that the information is correctly encapsulated in the TM. Regarding τ, the mean value of the difference between the measured τ and the value of τ specified by the user, was of approximately 50 μs. Therefore, the deviation in the TMW represented approximately the 50% of its duration. For this reason, it was necessary to propose a new solution to implement the inter-transmission time.

5.2 Second iteration: enhanced version of the multiple TM transmission in the master

In this section we find the solution proposed to solve the problems just identified for the first implemented solution of the SECSM in the master. Also, we find the results obtained from performing the corresponding test to evaluate its correctness.

5.2.1 Solution proposal and implementation

After studying the first implementation of the mechanism in the master, we concluded that the difference between the measured τ and the desired one was caused by the

function used to sleep the master between the transmission of the replicas. This was due to the fact that the process had to be awoken by the OS when the transmission moment was reached, which can introduce non-deterministic delays in the moment of the transmission, causing τ to be greater than specified.

Since the system has to be reactive, it is important to avoid the usage of functions that could prevent it from having such behaviour, as it is the case when the process is asleep. For this reason, we decided to use busy wait to implement this part of the mechanism. This way we can reduce the fluctuations of the real τ. Note that by using a busy wait to carry out the implementation we force the CPU to be busy between the replica transmissions. However, since during the TM window the master is only in charge of sending the TM replicas, the busy wait does not prevent other tasks from being executed.

Let consider the same scenario described in Section 5.1. Now after calculating $\beta(j,c)$ and constructing the TM $m_{j,c}$, the process gets in a loop where it constantly timestamps the current moment and compares it with $\beta(j,c)$. When the current moment is greater or equal to $\beta(j,c)$, $m_{j,c}$ the TM is sent. Next we find the pseudo-code for this implementation:

```
function dispatcher is
    loop
        sleep_until_next_EC();
        c++;
        for i in k do
            m(i,c) = construct_TM();
            now = timestamp();
            while now < tx(i,c) do
                now = timestamp();
            end while
            transmit(m(i,c));
            tx(i+1,c) = tx(i,c) + τ;
        end for
    end loop
end function
```

5.2.2 Testing

To test the accuracy of the real inter-transmission time reached by this new solution we used the test and the setup described in the previous Section. The results obtained by the test in this case were suitable, as the mean deviation of the measured τ only represented approximately the 0.5% of its desired duration.

CHAPTER 6

IMPLEMENTATION AND VALIDATION OF THE SECSM IN THE SLAVES

6.1 First iteration: preliminary version of the redundancy management in the slaves

As explained in Section2.3, in FTTRS the slaves have to manage the TM redundancy and determine the start of the SW. More specifically, the goal of this management is that all the slaves attached to the network have the same vision of the EC –that is, that all slaves consistently calculate the beginning of the following EC–, even in presence of transient faults in the channel.

This management is the part of the SECSM to be implemented at the slaves. In this section we explain the solutions proposed to implement it, as well as the setup and the results of the tests carried out to assess their correctness.

6.1.1 Solution proposal and implementation

In FTTRS the SW does not start when a TM is received, but when the TM window ends. This instant is called *TMW expiration time* and it is the moment when the slave dispatcher has to be launched. When a slave s receives a TM replica with sequence number i in EC c it has to calculate the TM window expiration time as it is indicated in Equation 2.1. In order to do so, when a TM is received, the process timestamps the moment of the reception and τ and the TM sequence number are extracted from its header.

In FTTRS masters also use the TM window to synchronize themselves [17]. Thus, since the masters are more synchronized at the end of the TM window than they are at the beginning, the later the TM is received, the most accurate the synchronization between slaves will be. For this reason, the TM window expiration time, $\alpha_s(k,c)$ (Section 2.3), is recalculated every time the slave receives a new TM replica within the EC c.

As has been said, in a non-faulty scenario, every slave will receive several TM replicas in each TM window. For this reason the `eth_filter` function can not be blocked when a TM replica is received, otherwise just the first TM replica received by a slave in each EC would be processed and the rest would be lost.

We first considered the possibility of using a timer that would expire when reaching the TMW expiration time. We decided that the API that we would use for implementing would be POSIX Timers. POSIX implements per-process interval timers which use signals to notify to the caller when they expire. We use a signal handler, which is a function described by the programmer, to define the actions that have to be carried out when the timer expires. POSIX also defines a series of *async-signal-safe* functions to handle signals. When an async-signal-safe functions is invoked by a signal handler the program shall behave as it is defined. This is the case of the function `recv()` used by the slave when receiving a message. Nevertheless, if an async-signal-safe function is interrupted by a signal handler then the function can either restart the call when the signal handler finishes its execution or fail with the error `EINTR`. This last one is the case for `recv()`, which fails with the error `EINTR` if it is interrupted by a signal handler before any data is available. For this reason, it was not possible to use timers to perform the implementation.

We finally decided to use a new thread to implement a busy wait. When the first TM replica of each EC is received by each one of the slaves, each slave creates a thread where it carries out the busy wait. If a slave receives a later TM replica within the same EC, the thread implementing the busy wait is cancelled and destroyed, the TM window expiration time is recalculated as seen before in Equation 2.1 and a new thread implementing the busy wait is created.

Next we find the pseudo-code for said solution :

```
function eth_filter is
    switch (type_of_msg)
        case TM:
            thread_cancel(dispatcher_launcher());
            thread_destroy(dispatcher_launcher());
            process_TM(m(i,c));
            calculate_EC_end();
            α(k,c) = now + (k − i)∗τ;
            create_thread(dispatcher_launcher());
            break;
        case ...
        default:
            error_msg(incorrect_msg_type);
            break;
    end switch
end function
```

Next we detail the dispatcher_launcher() function code, which is the function in charge of starting the SW:

```
function dipatcher_launcher is
    busy_wait(α(k,c));
    launch_dispatcher();
```

end function

Nevertheless, calculating the start of the synchronous window is not enough, in order to have the same vision of the EC slaves must consistently calculate when the EC ends. To this aim, each time a slave receives a TM replica it calculates the moment when the EC started.

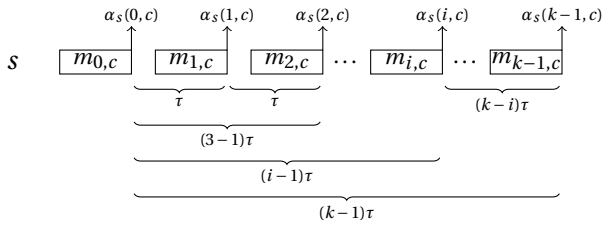


Figure 6.1: Vision of the EC start moment in a slave.

Figure 6.1 shows how the arrival of a TM replica in a certain EC is related to the start of that EC. We can see that, from the slave point of view, the start of an EC c coincides with the expected arrival time of the first TM replica belonging to that EC, this is, the start of EC c in slave s is $\alpha_s(1,c)$. Let consider that slave s receives the TM replica with sequence number i, where $i \in \mathbb{N}$ and $0 \leq i \leq k-1$ during EC c and let $\alpha_s(i,c)$ denote the expected arrival time of such replica, then the expected EC start moment would be calculated as follows:

$$\alpha_s(1,c) = \alpha_s(i,c) - i \cdot \tau \tag{6.1}$$

Once calculated the expected EC start moment the slave can calculate the end by adding the EC duration to that value. The EC duration is extracted from the TM header.

6.1.2 Testing

To assess the correctness of this implemented solution first we had to check whether the TMW expiration time was correctly calculated by the slaves or not. To this aim we designed a test that consisted in measuring the duration of the TMW in a non-faulty scenario. The expected duration for the TM window can be calculated as $(k-1) \cdot \tau$.

To measure the TM window duration in a given slave we calculate the time that elapses between the reception of the first TM replica and the moment when the slave launches the dispatcher, i.e. the moment when the TM window is considered to be finished. The instant when the TM is received is provided by the lower level reception primitives, while the moment when the dispatcher is launched is timestamped by the slave to latter calculate the difference.

Figure 6.2 shows the prototype used to perform this test. This prototype is conformed by a master and a slave node, both connected to a VDE switch by means of dedicated virtual links.

The duration of the TM window in a slave depends on the moment when it receives the different TM replicas from the master. The master sends the TM replicas with a constant inter-transmission time τ regardless of the number of nodes attached to the network. Thus, one slave is sufficient to carry out this test.

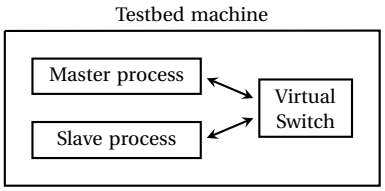


Figure 6.2: Prototype architecture with a single slave process and a virtual switch.

The results of the test revealed a lack of accuracy in this solution, which suffered from a lot of jitter, reaching values of the 30% of the EC duration. For this reason, we reconsidered the solution used. The description of the new solution can be found in the following Section.

6.2 Second iteration: enhanced version of the redundancy management in the slaves

In this section we describe the solution proposed to solve the problems of the previous solution. Also, we find the setup of the test carried out to assess the correctness of this new solution, as well as the results obtained from them.

6.2.1 Solution proposal and implementation

The imprecision of the first solution just explained was due to the fact that the thread that implemented the busy wait was created, cancelled and destroyed every time a TM replica was received by the slave.

Since POSIX threads are OS functions they can cause non-deterministic delays. In order to reduce as much as possible the jitter introduced by the thread usage, now the thread used to implement the busy wait is not cancelled, destroyed and created again every time a TM replica is received. Instead, only one thread is created in each EC. When a slave receives a TM replica it has to check if it is the first TM received in a certain EC c and, if so, the slave cancels and destroys the thread created in the EC $c-1$ and creates a new one after calculating the TM window expiration time. Thus, only one thread is created each EC for implementing the busy wait.

On the other hand, as seen in Section 2.3 slaves consider the TM window of a certain EC c to finish when the kth TM replica is received, so when a slave receives a TM this calculates the expected arrival time of the kth replica $\alpha_s(k,c)$. Nevertheless, the jitter in the transmission can cause the kth replica to reach the slave after $\alpha_s(k,c)$ has expired, that is, when the TM window is already over. When this happens, the slave must ignore the last replica as the dispatcher will have already been launched. This is done by checking the dispatcher status when a TM replica is received.

Next we find the pseudo-code of the new solution:

```
function eth_filter is
    switch (type_of_msg)
        case TM:
            if (c ≠ last_ec) then
                thread_cancel(dispatcher_launcher());
                thread_destroy(dispatcher_launcher());
            end if
            if not_dispatcher_launched then
                process_TM(m(i,c));
                calculate_EC_end();
                α(k,c) = now + (k − i)∗τ;
                if (c ≠ last_ec) then
                    create_thread(dispatcher_launcher());
                    last_ec=c;
                end if
            end if
            break;
        case ...
        default:
            error_msg(incorrect_msg_type);
            break;
    end switch
end function
```

The function dispatcher_launcher() is the same as the one described in the previous Section.

6.2.2 Testing

In order to ensure that the TMW duration is correct, we performed the test described in Subsection 6.1.2. The results obtained from the performance of such test –which measures the real TMW duration in a slave in a non-faulty scenario– show that this new solution reaches a suitable level of accuracy, as the deviation in the TMW is on the order of the 0.7% of the TMW duration.

On the other hand, in order to test the synchronization among the slaves reached by this solution it was necessary to describe a new test.

As previously discussed in Section 2.3, the reception of the TM replicas is the event used by the slaves to synchronize the start of the EC. The *EC offset* is a measure of how much the EC end moment differs between the slaves. As the synchronous and asynchronous windows duration is constant and is also the same for all the nodes in the network within each EC, we can calculate the EC offset as the difference between the Synchronous window start instants timestamped by two different slaves.

The prototype used to perform this test is the one shown in Figure 6.3. It is conformed by a master process and two slave processes attached to a VDE switch by means of virtual links.

As each node is connected to the switch through a dedicated link, the loss of a certain message in a slave does not necessarily affect to the reception of the same message in a different slave. Thus, we considered that using two slaves to test the slave

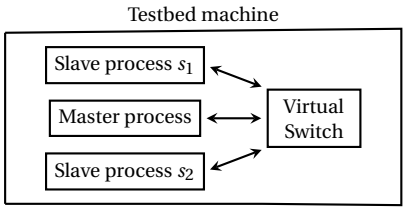


Figure 6.3: Prototype architecture with two slave processes and a virtual switch.

was a suitable solution, since the moment when a slave determines the EC to start depends on the master and not on the number of slaves.

The results obtained by performing this test revealed a lack of precision in the synchronization achieved by this strategy. Therefore, it was necessary to propose a further solution.

6.3 Third iteration: Turn Around Window (FTT-SE) implementation

It is important to note that the synchronization problems just explained are caused by a deficiency in the first specification/design of the SECSM. This means that the detection of this specification flaw is a contribution of this project. Moreover, in order to solve it, we proposed a modification of the specification, which is a novel contribution of this work.

In the following sections we describe the solution proposed by the candidate and its implementation, as well as the results of the test carried out to assess its correctness.

6.3.1 Solution proposal and implementation

The imprecision in the slave synchronization was due to the fact that slaves had no time to process the last TM replica, as the processing time of the TMs was not considered in the first specification of the mechanism when estimating the start of the SW. Therefore, we decided to include an idle time between the TMW and the SW, so that the nodes that receive the kth replica within the TMW have enough time to process it. FTT-SE uses a similar solution and calls this idle time the TAW, as can be seen in Section 2.2, so we decided to keep this terminology.

Figure 6.4 shows the new structure of the FTTRS EC.

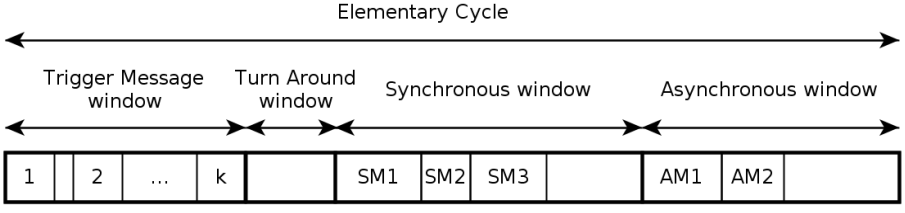


Figure 6.4: EC structure in the new FTTRS specification.

Note that now the expected arrival time of the kth replica does not coincide with the start of the SW, but with the start of the TAW. Next we will describe in detail how the TAW has been implemented.

When a slave receives a TM replica with sequence number i it estimates the EC start moment as seen in Equation 6.1 and saves it in a database to which the slave dispatcher is able to access in order to read it and determine the end of the EC. This same value is also used by the slave dispatcher to estimate the SW start. Let $\alpha_s(1,c)$ denote the expected arrival time of the first TM replica to slave s in EC c. From the slave point of view this moment coincides with the start of EC c. To this moment the slave dispatcher must add the TMW and the TAW length to estimate the EC end.

$$TMW_{length} = (k-1)\cdot\tau \tag{6.2}$$

Since the slave dispatcher has not direct access to the information carried in the TMs, the TMW length is calculated in the eth_filter file as seen in Equation 6.2 and then this value is passed to the dispatcher as a parameter of the function used to launch the TAW. Finally, the estimated SW start is calculated in the dispatcher as specified in Equation 6.3.

$$SW_{start} = \alpha_s(1,c) + TMW_{length} + TAW_{length} \tag{6.3}$$

The TAW length is a configuration parameter that can be changed depending on the dynamic of the system. Therefore, its duration can be set to 0, eliminating the TAW if it is considered to be not necessary.

6.3.2 Testing

Once finished the implementation of the TAW it was necessary to ensure that it was a suitable solution to the problems in the slaves synchronization. First, we performed a test to measure the accuracy in the synchronization among slaves in a non-faulty scenario for different TAW lengths. This test was performed using the test setup described in the previous Section. As it is later discussed in Section 7.3 the TAW improves the accuracy of the synchronization and, therefore, we considered it to be a suitable solution.

Also we performed an exhaustive verification of the mechanism and its implementation. We used this test to assess the robustness of the SECSM in front of TM losses. We also wanted to assess the correctness of the implementation. We used software implemented fault injection (SWIFI) to inject the transient faults in the channel. The SWIFI code has been added to the eth_filter file. This code forces the slaves to ignore some TM replicas in order to simulate every possible combination where up to $k-1$ TM are lost. Thus, we test every possible scenario where the mechanism is capable of synchronizing the slaves in front of transient faults. The number of TM loss combinations is given by

$$\left(\sum_{e=0}^{k-1}\binom{k}{e}\right)^{n}, \tag{6.4}$$

where k is the number of TMs per EC, e is the number of lost TMs, and n is the number of slaves attached to the network.

The SWIFI code uses a function that returns every possible combination as an array of booleans. When a TM replica is received its sequence number is used to index the array position where it is indicated whether the TM must be processed, '1', or ignored, '0'.

Two different setups were used to carry out this test. The first one was performed with the prototype explained in Subsection 6.2.2. It is conformed by a master and two slave processes attached to a VDE switch. This configuration allowed us for comparing the results obtained from this test to the ones obtained in a non-faulty scenario.

The second setup is the one seen in Figure 6.5 and differs from the first one in the network configuration. This setup is conformed by a master process and two slave processes, all executed in the same machine, attached to a *commercial off-the-shelf* (COTS) switch through physical Ethernet interfaces. Each process is attached to a different Ethernet interface of the machine where the test is executed. This prototype allowed us for testing the functioning of the implementation when using physical network components.

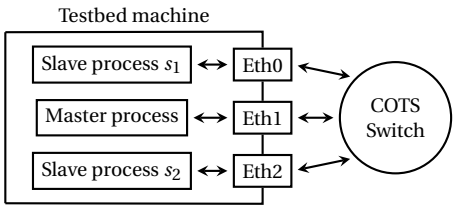


Figure 6.5: Prototype architecture with two slave processes and a COTS switch.

The results obtained by performing these tests were considered to be suitable in terms of synchronization and showed a right behaviour of the mechanism in front of transient faults. A deeper discussion of the results can be found in Chapter 7.

CHAPTER 7

EVALUATION OF THE INTEGRATED PROTOTYPE

In this section we describe the results obtained from the tests carried out on the complete prototype of the SECSM, i.e. on the prototype that results from integrating the different mechanisms implemented during the *implementation and validation* phase.

Concretely, we have tested the behaviour of each part of the mechanism when working together in the integrated prototypes. Note that part of the integration was actually done in the implementation and validation phase, since in order to validate the SECSM mechanisms located in the slaves it was necessary to test them in conjunction with the SECSM mechanisms of the master. This also means that the test setups used in the current section are the two last ones presented in Chapter 6, namely the one that uses a virtual switch (Figure 6.5) and the one that uses a COTS-based one (Figure 6.3.

7.1 TM replica transmission

The first tests we conducted were devoted to checking that the content of the TM is correct, as well as to measuring the time that elapses between the transmission of the different TM replicas that belong to the same EC. We used the prototype with the virtual switch to carry out this test (Figure 6.5).

In order to verify the content of the TM we captured the traffic in the master link with Wireshark. We observed that the content of the TM was correct, as shown in the frame example of Figure 7.1. This frame corresponds to the first TM replica in the EC with sequence number 188, with a k of 4, a value of τ of 100 μs and an EC duration of 1 ms.

The content of the frame is shown as a stream of bytes expressed as hexadecimal values, thus two digits represent one byte of information. As it can be seen, a frame is divided in four different parts:

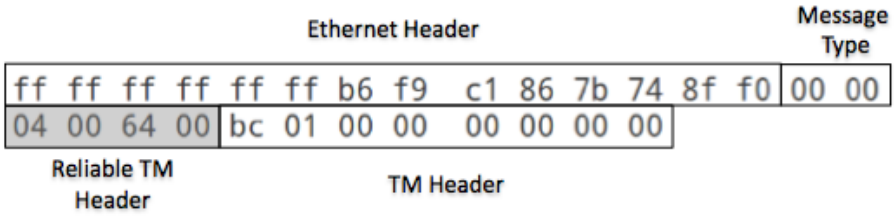


Figure 7.1: Wireshark capture of a TM in the master link.

- First, the Ethernet header which contains the destination and source addresses, as well as the Ethernet frame type which is 8ff0 for FTT frames.
- Next, the FTT message type, the value of which is 0 corresponding to a TM.
- The reliable TM header, which contains the information related to the SECSM, namely:
  - 1 byte for k. In this case its value is 4.
  - 1 byte for the TM sequence number, the value of which is 0, corresponding to the first TM replica of the EC.
  - 2 bytes for τ. The byte ordering schema of the machine causes that the fields of two or more bytes are reverted. For this reason τ is coded as 64 00, which corresponds to 64_{16}, 100_{10}.
- Finally, the TM header contains five fields. First the EC sequence number, which is BC_{16} and corresponds to 188_{10}. Next the EC duration in ms that, as said before, is 1. Finally the number of synchronous, asynchronous and signalling messages. Note that, this last 6 bytes are 0 due to the fact that the TM does not carry any scheduling information.

As concerns the part of the tests devoted to assessing the time between the transmission of different TM replicas of an EC, we measured the real duration of the TM inter-transmission time. This is done by timestamping the moment of the transmission of one replica and the following one and calculating the difference between both transmissions. Then, we calculated the difference with respect to the specified τ.

This test was executed 1000 times, which corresponds to 225000 ECs. Thus the number of samples of this test was 675000 since the number of inter-transmissions in each EC is 3. The following table shows the maximum, the mean and the standard deviation of the measured differences with respect to τ.

Specified τ (μs)	Real τ		
	max (μs)	mean (μs)	Std. dev. (μs)
100	5.984	0.349	0.125

Table 7.1: Measured difference between the real and the expected τ.

Note that the maximum deviation of the real inter-transmission time just represents the 5.98% of the desired τ and only the 0.59% of the EC size. Moreover, the value of

Expected TM window (μs)	Measured difference		
	max (μs)	mean (μs)	Std. dev. (μs)
300	61.408	0.89	2.789

Table 7.2: Measured difference between the real and the expected TMW duration.

the mean and the standard deviation show that there is a reduced number of peaks, this is, the values close to the maximum deviation are not common. The results of this test demonstrate that the design and implementation of the part of the SECSM located in the master are suitable for the EC sizes considered in this project, which are representative of a wide range of control applications.

7.2 TM window duration in the slave nodes

The second set of tests we carried out were aimed at measuring the real duration of the TMW observed in the slaves and, then, to compare them with the expected TMW duration. For that we used the setup with the virtual switch (Figure 6.5).

Theoretically, the TMW lasts $(k-1)\cdot\tau$ time units. Thus, for a k of 4 and a τ of 100 μs, the expected duration of the window is 300 μs. Table 7.2 shows the maximum, the mean and the standard deviation of the measured differences between the expected TMW duration and the observed one.

As can be seen there, the maximum difference between the expected TMW duration and the real one is 61.408 μs. This value just represents the 6.1% of the EC duration and, thus, it is considered to be a suitable result. Moreover, the mean and standard deviation show that in most cases the results obtained by the implementation are good.

Additionally, Figure 7.2 shows the TMW duration for the different samples. Positive and negative values respectively correspond to samples in which the TMW is greater and lower than the expected value.

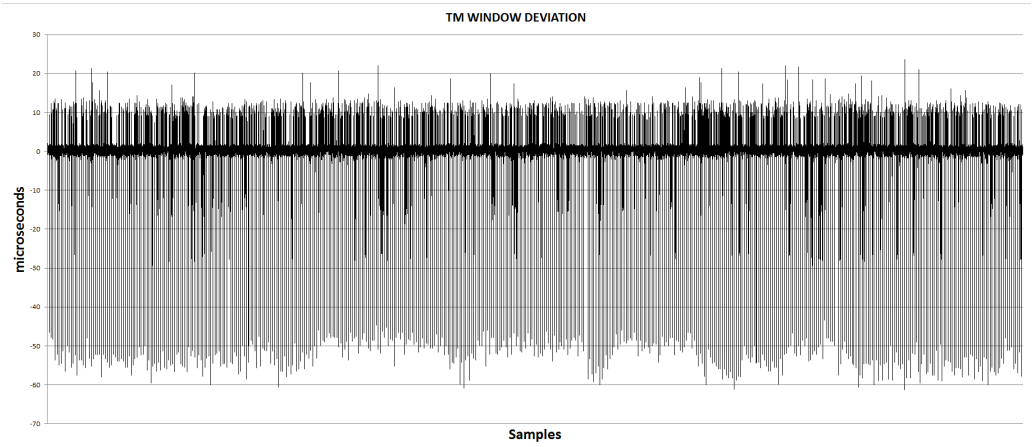


Figure 7.2: Deviation in the TMW duration in μs.

As observed in this figure, the greater peaks correspond to cases in which the TMW duration is less than it should. This is due to the fact that the reception of the first

TM replica is sometimes delayed, while the other replicas reach the slave in the right moment. The reception delay of the first TM is caused by the indeterministic behaviour of the underlaying software components and, therefore, solving this issue is out of the scope of this work.

Fortunately, the mean and the standard deviation of the results (Table 7.2) indicate that this behaviour is very unlikely. Furthermore, the maximum measured difference just represents the 6.1% of the EC duration. In any case, it is noteworthy that the reception of any TM replica forces the slave to recalculate the TMW end moment. This means that a delay in the first TM can only negatively affect the synchronization among the slaves when a subset of slaves do receive just the first TM replica and do not receive the other ones. Therefore, the probability with which slaves can lose the synchronism because of the reception delay of a TM is very low, or even negligible when an appropriate value of k is used. In conclusion, the results of these tests are considered acceptable for a wide range of control applications.

7.3 Slaves synchronization in a non-faulty scenario

The last set of tests we carried out were devoted to measure the divergence in the predicted SW start among two slaves, that is, the EC Offset. Note that these test are the ones we pointed out in Section 6.3.2 when explaining the validation of the TAW mechanism. Particularly, in the current section we show the results of the measured EC Offset when faults do not occur and all TM replicas are received by the slaves. We used the virtual switch prototype to carry out these tests.

In order to be as general as possible we defined different values for the TAW duration. Specifically we performed the tests with TAW values of 0 μs and 50 μs.

Figure 7.3 shows the results obtained from measuring the EC Offset with a TAW duration value of 0 μs. As it can be seen, the SW start suffers from a lot of jitter. This effect is caused by the jitter with which the master itself transmits different TM replicas (variations in the transmission times are due to the indeterministic behaviour of the master underlaying software components). To better understand why this transmission jitter provokes jitter in the SW start, note that the transmission jitter can cause that the last TM replica, i.e. the kth replica, reaches some slaves within the TMW and other slaves when the TMW has already expired. If this happens, only the slaves that do receive the replica within the TMW need extra time to process it and, thus, they delay the SW start with respect to the other slaves.

Figure 7.4 shows the results obtained when performing the same test with a TAW length of 50 μs. It can be observed that increasing the TAW in such a way leads to a more accurate synchronization.

The difference in the results of both configurations is due to the fact that in the second one all the slaves have enough time to process the kth TM replica.

Table 7.3 shows the maximum, the mean and the standard deviation of the measured EC Offset for both configurations.

The mean shows that the EC synchronization achieved is good for both configurations, just the 2% and 2.5% of the EC duration. In any case, results also show the advantages of the second configuration (the one in which TAW = 50). On the one hand, we can see that the standard deviation is five times smaller when TAW = 50 μs than

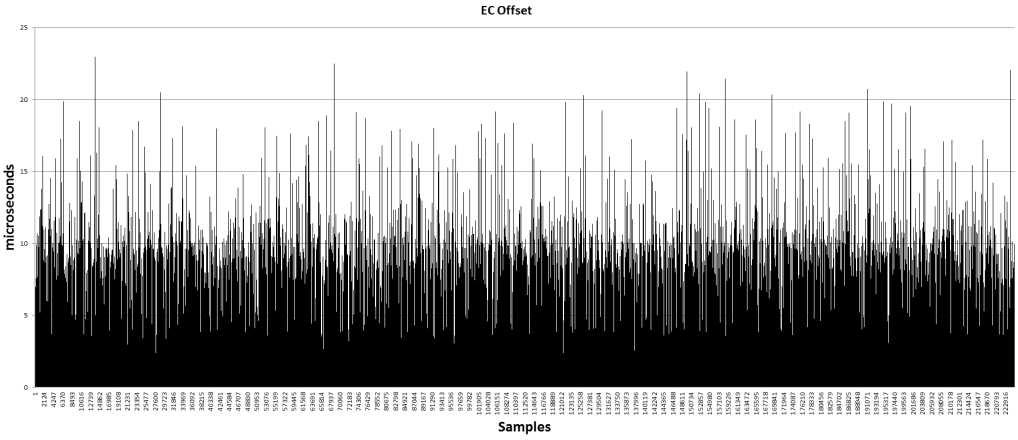


Figure 7.3: Measured EC Offset with a TAW duration of 0 μs in absence of faults.

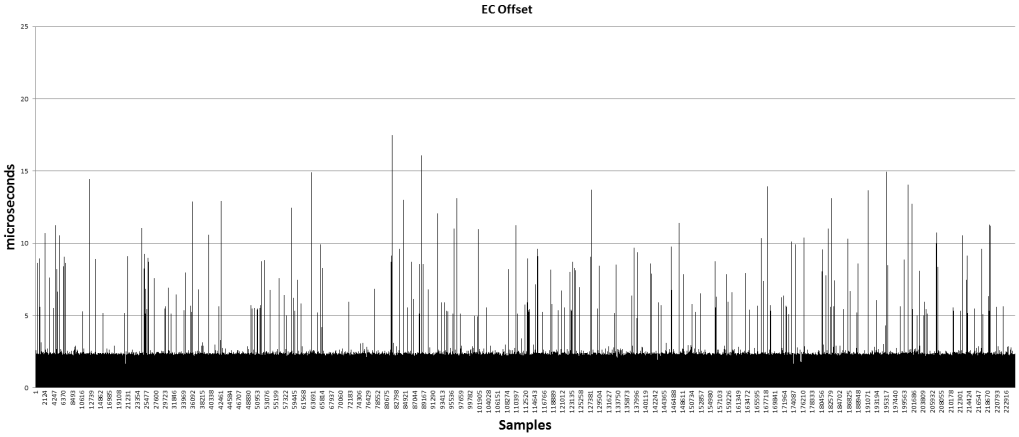


Figure 7.4: Measured EC Offset with a TAW duration of 50 μs in absence of faults.

Turn Around window (μs)	Measured EC Offset		
	max (μs)	mean (μs)	Std. dev. (μs)
0	22.967	2.561	1.607
50	17.499	2.044	0.313

Table 7.3: Measured EC offset values for TAW values of 0 and 50 μs in absence of faults.

when TAW = 0 μs. This shows that values of TAW greater than 0 helps in mitigating the effects derived from the jitter in the transmission of the TM. On the other hand, we can observe that the second configuration reduces the maximum EC Offset, from 22.967 to 17.499 μs.

7.4 Slaves synchronization in scenarios involving transient faults

Next we describe the results of the test we performed to assess the EC Offset in scenarios involving faults that provoke the loss of TM replicas. As already pointed out in Section 6.3.2 in order to exhaustively check the correctness of the SECSM in these scenarios, we included within the slaves a software code that leads them to ignore some TM replicas in every possible combination.

To perform this tests we used both setups, the one with a virtual switch (Figure 6.5) and the one with a COTS switch (Figure 6.3).

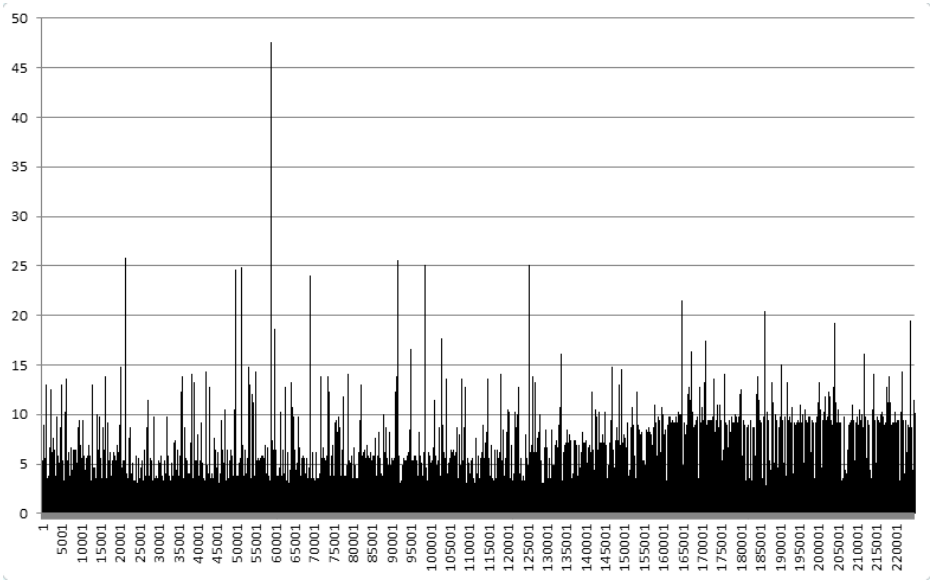


Figure 7.5: Measured EC Offset with virtual switch in presence of transient faults. The horizontal axe shows the number of samples. The vertical axe shows the EC Offset in microseconds.

Figure 7.5 shows the results obtained when using the virtual switch. If we compare these results with the ones shown in Figure 7.3, which were obtained in a non-faulty scenario with a virtual switch, we can see that the loss of some TM replicas causes the EC synchronization to be less accurate.

To better understand this loss of accuracy, recall that in Section 7.2 we explained that in some cases the reception of the first TM replica of an EC is delayed (due to the non-deterministic behaviour of software components). In principle, slaves can solve this problem by recalculating the SW start with the reception of later TM replicas. However, the omission of some TMs prevents the slaves from recalculating the SW start. If this occurs the slaves cannot resynchronize, which leads to a higher deviation in the EC synchronization.

Figure 7.6 shows the measured EC Offset obtained when using a COTS switch. It should be noted that not only the maximum EC Offset is greater than in the logical implementation, but also that there are more peaks with values over 50 μs. This can be explained by the fact that physical COTS switches present a non-constant forwarding

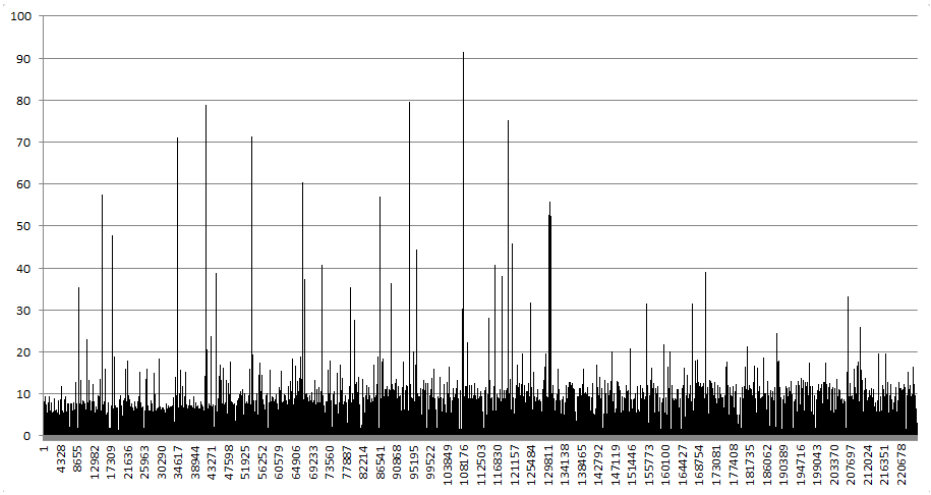


Figure 7.6: Measured EC Offset with COTS switch in presence of transient faults. The horizontal axe shows the number of samples. The vertical axe shows the EC Offset in microseconds.

delay greater than the one of virtual switches. In fact, since during the TMW the TM replicas sent by the master are the only traffic in the network, they are not queued in the output ports of the switch. Thus, the non-constant delay is a characteristic of the switching physical mechanisms themselves.

Configuration	Measured EC Offset		
	max (μs)	mean (μs)	Std. dev. (μs)
Viratual Switch	47.62	1.94	0.84
COTS Switch	91.37	0.69	1.36

Table 7.4: Measured EC offset values for a virtual and a COTS switch in presence of transient faults.

Table 7.4 show the maximum, mean and standard deviation of the measured EC Offset for a virtual and a COTS switch. As discussed in Section 7.3, the first TM replica is sometimes delayed as a consequence of the non-deterministic behaviour of the software components that constitute the virtual network. For this reason, the mean of the measured EC offset is worse when using the virtual switch than when using the COTS one.

In principle, the mean and the standard deviation are acceptable for both configurations. Nevertheless, we can observe that the maximum value of the measured EC offset is significant, i.e. the 5% and 10% of the EC duration respectively. Since in FTTRS the end of the EC is the deadline for the round-base communication, this deviation in the synchronization can prevent the slaves from sending some messages in the right EC.

Even though, the problems that affect the correct operation of the implementation are a consequence of the software-implemented platform and, therefore, they are out of the scope of this project.

CHAPTER 8

CONCLUSIONS

In this chapter we first present a summary of the work that has been carried out in this bachelor project, as well as of the conclusions reached during its development. Moreover, for future work, we suggest some task that could be carried out within the FT4FTT project [8]. Finally, we include a series of considerations about the learning process of the author of this bachelor thesis.

8.1 Summary

This project consisted in the implementation and validation of the slave elementary cycle synchronization mechanism (SECSM), proposed as part of the FT4FTT project. FT4FTT faces the construction of a highly-dependable and flexible communication infrastructure based on the FTT-Ethernet protocol.

For this implementation we had as point of departure a first specification of the system [10]. Moreover, we also had a software implementation (middleware) of the FTT-SE communication protocol [7] [11], which has been used as the development platform for this project. Specifically, we placed the SECSM within the core layer of the FTT-SE middleware in the master and in the slaves. Figure 4.1 shows as circles the specific parts of the middleware that have been modified.

Regarding the dispatcher of the master, we performed various modifications. First, since both the duration of the TMW and the duration of the EC are parameters decided by the user, our implementation has to ensure that the TMW fits within the EC, i.e., the slave did not make any mistakes. Also, we added the code to perform several transmissions of the TM in each EC.

Concerning the slave software, the Ethernet filter was modified in order to manage the redundancy of the TM, as well as to enable each slave to calculate the start moments of the synchronous window and the following EC from the set of TM replicas received. Moreover, we wanted to inject transient faults in the channel to make an exhaustive verification of the mechanism. In order to do that, we chose a software implemented

approach and we added some code so as to omit the processing of some TM replicas.

Also, in order to determine the start of the synchronous window and the following EC the slaves need information regarding the number of TM replicas transmitted and the time between their transmissions. Thus, since this information is set in the master, the TM header had to be modified in order to convey said information.

Once finished the implementation of the original specification of the slave EC synchronization mechanism, we carried out a series of tests in order to assess the correctness of the specification and the implementation. The results of these tests showed that the implementation offers a low level of accuracy in the synchronization. That is, the difference between the EC start moment predicted by each slave was significant.

Note that the first specification of the slave EC synchronization mechanism considered the reception of the last TM replica to be the start moment of the synchronous window. Thus, as the time that the slaves need to process the TM is not negligible, the calculation of the synchronous window start moment was incorrect.

To solve this problem we proposed the inclusion of the TAW in the specification of the slave EC synchronization mechanism. This way the nodes have enough time to process the TM before the synchronous window starts. The TAW was not implemented in the initial development platform even though it was part of the FTT-SE protocol specification. Therefore, we modified the slave dispatcher in order to add it.

After finishing the integration of all the parts of the mechanism we carried out various tests. We measure how different aspects related to imprecision of the implementation of the mechanisms affect the synchronization among the slaves. Basically, we assessed the precision with which the slaves have the same view of the start of the synchronous window within the EC, depending on aspects like the variation in the size of the window in which the TM is transmitted (TMW), variation in the amount of time during which the slaves process the TMs (TAW), the loss of any combination of TMs due to transient faults, and the jitter introduced by the non-deterministic behaviour of software and the switching logic.

The results obtained in this assessment indicate that the SECSM implementation is suitable for most control applications, even though it could be problematic for control applications in which high sampling rates demand ECs of 1 ms or less.

Finally, it is important to highlight that the author of this work has co-authored a published short paper which describes the main results of this bachelor thesis. The other co-authors of said paper were David Gessner, Alberto Ballesteros, Manuel Barranco and Julián Proenza. Next we find the citation of this publication:

- Gessner, D., Álvarez, I., Ballesteros, A., Barranco, M., Proenza, J., *Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet.* In Proc. 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), September 16-19, 2014, Barcelona, Spain.

The complete article can be found in Annex B of this document.

8.2 Future work

This bachelor thesis achieves the goals that we established at the beginning of its development. Nevertheless, during the project we faced a series of time and material restrictions that prevented us from implementing a completely realistic prototype of the slave elementary cycle synchronization mechanism (SECSM).

Although the prototype allowed us to assess how the precision of the SECSM is affected by several relevant causes, there are still some sources of desynchronization that could not be tested. For example, all the slaves in our prototypes are processes executed in the same machine and, thus, we could not assess the impact produced by the deviation of the slave clocks during the EC.

Also, note that the prototype proposed in this project uses one master, whereas the FTTRS[9] specification includes two of them. Moreover, although we used Xenomai –which is a Linux kernel that offers real-time features– the existing software implementation of FTT-SE we were provided with was not adapted to work with this kernel. Thus, we could not take full advantage of the real-time features offered by Xenomai.

This project constitutes a relevant first step towards a complete verification of the SECSM for the FTTRS infrastructure. In this sense, given the limitations just pointed out, we still need to carry out a series of tasks as future work within the FT4FTT project in order to construct a completely realistic prototype of the SECSM within the FTTRS architecture. Some of the most relevant task are the following ones:

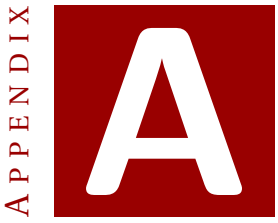
- Implement each slave of the distributed embedded system in a different machine.
- Integrate the SECSM proposed in the current bachelor project with the mechanism that synchronizes the replicated masters [17].
- Adapt the FTT-SE code to work with the real-time features offered by Xenomai.

8.3 Considerations about the learning process

The development of this project also contributed to increase the personal knowledge, experiences and skills of the author of this work. Next we mention the most remarkable benefits that were obtained along this project:

- Increase the knowledge achieved during these studies, e.g., regarding real-time systems and the Ethernet protocol, among other subjects. Also, learn new concepts and work in new fields of study, such as fault tolerance and real time communications.
- Increase the understanding of the working environment and the practical knowledge of the programming language. Moreover, working with an unknown development platform required studying and using new components, e.g., virtual distributed Ethernet switches.
- Plan and develop a project with a size considerably larger than those carried out during these bachelor studies..

- Acquire of skills related to the writing of scientific articles during the participation in the creation of a short paper[18].



SOURCE CODE

In this chapter we can find the code of the different files that have been modified during this project. These files correspond to the ones that were previously identified in Chapter 4. As previously highlighted, we took as starting point of this project a software implementation of the FTT-SE protocol [7] [11]. This means we did not develop the complete code shown in the following sections. Therefore, we decided to differentiate both codes by putting the code written during this project in a frame.

A.1 Master dispatcher

```
/*****************************************************************************
 * M_Dispatcher.c:
 *****************************************************************************
 * Copyright (C) 2006-2012 the FTT-SE team.
 *
 * Author: Ricardo Marau <marau at fe.up.pt>
 *
 * FTT-SE is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  FTT-SE is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with FTT-SE.  If not, see <http://www.gnu.org/licenses/>.
 *****************************************************************************/

#include "Ports.h"

#include "ftt-global.h"
//#include "SRDB.h"
#include "M_ECreg.h"
#include "M_Scheduler.h"
```

```
//#include "math_tools.h"
#include "ethernet.layer.h"
#include "M_SW_add_table.h"
//
#include "S_ftt-core.layer_embed.h"



//#define TRIGGER_DECODE
//#define BE_POLITE
////#define DISABLE_TX

static rtl_pthread_t ftt_dispatcher_th;
static FTT_SEMAPHORE start_dispatcher;

static unsigned char M_MAC[6];
static unsigned long M_ec_period=0; //us
```

```
static unsigned char k;
static unsigned short tm_interarrival;
```

```
void *ftt_dispatcher(void *t)
/*****************************************************************************/
/*!
\brief This task is invoqued periodically (at the beginning of each EC)

  This task is invoqued periodically (at the beginning of each EC).\n

  - scans current EC (sched_ec),\n
  - builds respective master message and,\n
  - send it to the bus.

******************************************************************************/
{
  unsigned short i;

  unsigned char *tmsg;

  FTT_PKT_TM_HEADERS *tm_pkt; /* Pointer to TM contents */
  TM_SYNC_MSG_INDEX *tm_sm_element;
  TM_ASYNC_MSG_INDEX *tm_am_element;
  TM_SIG_ACK_INDEX *tm_sig_ack_element;

  unsigned short tm_datalength; /* Data lenght of the TM frame */

  struct rtl_timespec next_activation; //used to time activations
```

```
  unsigned char EC_seqno_temp; //EC sequence number
  unsigned char tm_trans_seq_number = 0; //TM sequence number within the same EC
  int ret;

  struct rtl_timespec next_tm_activation; //used to time activations of the TM
  struct rtl_timespec temp;
  struct rtl_timespec start;
  struct rtl_timespec resul;
```

```
  unsigned short ret_node_id;
  unsigned char ret_seq_no;
  unsigned short num_sig_acks;

  /* Wait for start */
  if ( FTT_SEMAPHORE_wait(&start_dispatcher) < 0 )
    return NULL;
```

```
//  PRINT_MSG("Dispatcher Started");

  /* get the current time and setup for the first tick */
  rtl_clock_gettime( RTL_CLOCK_REALTIME, &next_activation );
  while (1)
  {
    /* set the period */
    rtl_timespec_add_ns( &next_activation, 1000*M_ec_period );


    /* sleep */
    rtl_clock_nanosleep( RTL_CLOCK_REALTIME, RTL_TIMER_ABSTIME,
        &next_activation, NULL);
//    {
//      struct rtl_timespec timestamp;
//      rtl_clock_gettime( RTL_CLOCK_REALTIME, &timestamp );
//      PRINT_MSG("Dispatcher end: %lld", (timestamp.tv_sec*1000 +
    (timestamp.tv_nsec/1000000)));
//    }

    ECreg_SwapPlans();
    EC_seqno_temp = (unsigned char)EC_Get_and_Inc_seqno();
#ifdef BE_POLITE
    PRINT_MSG("ftt_dispatcher:␣Hello...!");
#endif
```

```
    for ( tm_trans_seq_number = 0; tm_trans_seq_number < k;
        tm_trans_seq_number++ ){
```

```
      /* Check  EC status */
      switch( disp_ec->status ){
        case OUT_OF_DATE:        /* Starting up :  do nothing */
        case UPDATED:            /* OK. Follow down to normal plan dispatching
            */
          //PRINT_MSG("ftt_dispatcher fine");
          break;
        case MISSED_DEADLINE:    /* Missed deadline detected by scheduler,
            terminate */
          PRINT_MSG("ftt_dispatcher␣Fatal␣error:␣missed␣deadline␣in␣synchronous␣
              message");
          return NULL;
          break;
        case BUSY_SCHED:         /* Scheduler did not build plan in time,
            terminate */
          PRINT_MSG("BUSY");
        case DISPATCHED:         /* Scheduler has not been started
                                 */
          PRINT_MSG("ftt_dispatcher␣Fatal␣error:␣Scheduler␣has␣not␣finished␣in␣
              time");
          return NULL;
          break;
        default:
          PRINT_MSG("ftt_dispatcher␣Fatal␣error:␣Invalid␣EC␣Status␣(Strange␣
              error!)");
          break;
      }

      if ( ETH_L_tx_reserve_buffer( &tmsg ) < 0 ){
        ERROR_MSG("Problems␣getting␣a␣new␣Tx␣buffer␣-␣Progamming␣error␣");
      }

      tm_pkt = (FTT_PKT_TM_HEADERS *)tmsg;

      /* The source and destination address */
      ETH_L_Str2Addr((unsigned char *)"FF:FF:FF:FF:FF:FF",
          tm_pkt->eth_header.eth_dest );
```

```
      ETH_L_CopyMAC_to_from(tm_pkt->eth_header.eth_src, M_MAC );

      /* The frame type */
      tm_pkt->eth_header.eth_type = htons ( ETH_FTT_TYPE );
      tm_pkt->pkt_header.type = htons ( FTT_MST_MSG );

      /*The reliable TM data*/
      tm_pkt->tm_reliable_header.num_tms_per_ec = k;
      tm_pkt->tm_reliable_header.tm_sequence_number = tm_trans_seq_number;
      tm_pkt->tm_reliable_header.tm_interarrival_time = tm_interarrival;

      /* Given disp_ec, we can translate it into the TM */
      tm_pkt->tm_header.seq_no = EC_seqno_temp;
      tm_pkt->tm_header.ec_time_ms = M_ec_period/1000;
      tm_pkt->tm_header.nsm   = htons((unsigned short)disp_ec->sm.n_mesgs);
      tm_pkt->tm_header.nam   = htons((unsigned short)disp_ec->am.n_mesgs);

#ifdef TRIGGER_DECODE
      PRINT_MSG("TM: nsm=%2d nam=%2d",disp_ec->sm.n_mesgs,disp_ec->am.n_mesgs);
#endif
      /* Synchronous message data */
      tm_sm_element = (TM_SYNC_MSG_INDEX *)(tm_pkt->data);
      for( i=0; i < disp_ec->sm.n_mesgs; i++ ){
        tm_sm_element[i].SMesgId = htons( (unsigned
            short)(disp_ec->sm.msgs[i].id) );
        tm_sm_element[i].SMesg_frag_no = htons(
            disp_ec->sm.msgs[i].n_fragmentation );
        tm_sm_element[i].source_nodeID = disp_ec->sm.msgs[i].source_nodeID;
#ifdef TRIGGER_DECODE
        PRINT_MSG("      S_id:%04x   frag:%d  src:%d", disp_ec->sm.msgs[i].id,
            disp_ec->sm.msgs[i].n_fragmentation,
            disp_ec->sm.msgs[i].source_nodeID);
#endif
      }

      /* Asynchronous message data */
      tm_am_element = (TM_ASYNC_MSG_INDEX *)( tm_pkt->data + (
          disp_ec->sm.n_mesgs*sizeof(TM_SYNC_MSG_INDEX)) );
      for( i=0; i < disp_ec->am.n_mesgs; i++ ){
        tm_am_element[i].AMesgId = htons( (unsigned
            short)(disp_ec->am.msgs[i].id) );
        tm_am_element[i].AMesg_frag_no =
            htons(disp_ec->am.msgs[i].n_fragmentation);
        tm_am_element[i].source_nodeID = disp_ec->am.msgs[i].source_nodeID;
#ifdef TRIGGER_DECODE
        PRINT_MSG("      A_id:%04x   frag:%d  src:%d", disp_ec->am.msgs[i].id,
            disp_ec->am.msgs[i].n_fragmentation,
            disp_ec->am.msgs[i].source_nodeID);
#endif

      }

      /* Notify Signaling !ack */
      num_sig_acks = 0;

      tm_sig_ack_element = (TM_SIG_ACK_INDEX *)( tm_pkt->data +
          (disp_ec->sm.n_mesgs*sizeof(TM_SYNC_MSG_INDEX)) +
          (disp_ec->am.n_mesgs*sizeof(TM_ASYNC_MSG_INDEX)) );
      tm_pkt->tm_header.nsa = 0;

      while (EC_pull_Signaling_notification( &ret_node_id, &ret_seq_no ) == 0 )
          {
        tm_sig_ack_element->nodeID = htons ( ret_node_id );
        tm_sig_ack_element->sig_seqno = ret_seq_no;
        tm_sig_ack_element++;
```

```
        num_sig_acks++;
      }
      tm_pkt->tm_header.nsa    = htons((unsigned short)num_sig_acks);

      /*****************************/
      /* Prepare the transmission */
      /*****************************/
      tm_datalength = FTT_PKT_TM_LOGICALSIZE_B_from_no(disp_ec->sm.n_mesgs,
          disp_ec->am.n_mesgs, num_sig_acks);
//    PRINT_MSG(" TM size:%d", tm_datalength);
//#define TRIGGER_PERIODIC
#ifdef TRIGGER_PERIODIC
#error hi
      /* set absolute delay to reduce the system jitter */

      struct rtl_timespec temp_activation; //used to time activations
      temp_activation = next_activation;

      timespec_add_ns( &temp_activation, 1000*30 );

      /* sleep */
      rtl_clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME, &temp_activation,
          NULL);

#endif
#ifndef DISABLE_TX
```

```
      /* Send the message */
      if (tm_trans_seq_number > 0){

        /* Sleep until next TM transmission */
        rtl_clock_gettime( RTL_CLOCK_REALTIME, &temp );
        while (rtl_timespec_gt( next_tm_activation,temp )){
          rtl_clock_gettime( RTL_CLOCK_REALTIME, &temp );
        }
      }

      rtl_clock_gettime( RTL_CLOCK_REALTIME, &next_tm_activation );
      temp = next_tm_activation;
```

```
      if ( (ret = ETH_L_tx_send_messagebuffer(NULL, tm_datalength)) < 0 ){
        ERROR_MSG("Message not sent. ETH_L_tx_send_messagebuffer returned: %d",
            ret);
      }
```

```
      /* set the period */
      rtl_timespec_add_ns( &next_tm_activation, tm_interarrival*1000 );

      if (tm_trans_seq_number > 0){
        resul = rtl_timespec_sub(start, temp);
        printf("EC: %3d TM: %6llu\n", EC_seqno_temp, rtl_timespec_getns(resul));
      }
      start = temp;
    }
```

```
#endif
    /* Declare the table as dispatched */
    disp_ec->status = DISPATCHED;

/* We will simulate that the TM was received, right now. So, it has to be
    decoded and the
 * varibles that are produced here transmitted */
    struct rtl_timespec TM_tx_time;
```

```
    rtl_clock_gettime( RTL_CLOCK_REALTIME, &TM_tx_time );
    S_eth_filter( tmsg, tm_datalength, TM_tx_time );

    /* Activate scheduler() */
    WakeSheduler();

    /* Let's now downgrade the Node's connection aging time */
    M_SW_downall_lease_time();

  }//while 1
}

signed char M_DispatcherInit(unsigned char *mac_reg, unsigned long
    ec_period_reg, unsigned char tm_per_ec, unsigned short tm_inter_time )
{

  rtl_pthread_attr_t attr;
  struct rtl_sched_param sched_param;
  int ret;

  /* Store the MAC and EC_period in this module */
  ETH_L_CopyMAC_to_from( M_MAC, mac_reg );
  M_ec_period = ec_period_reg;

  /* Store Number of TM_per_EC and TM_interarrival_time */
  k = tm_per_ec;
  tm_interarrival = tm_inter_time;

  /*Check if k is at least 2 */
  if ( k < 2){
    ERROR_MSG("The␣number␣of␣transmissions␣of␣the␣TM␣must␣be␣at␣least␣two");
    return -3;
  }

  /*Check if there are too many transmissions of the TM */
  if ( ( k*tm_interarrival ) > M_ec_period){
    ERROR_MSG("The␣elementary␣cycle␣duration␣is␣too␣short␣for␣the␣given␣trigger␣
        message␣transmission␣window");
    return -3;
  }


  /* Init semaphore to start periodic dispatch task */
  if ( (ret=FTT_SEMAPHORE_init(&start_dispatcher, 0)) < 0 ){  //Initialization
      locked
    ERROR_MSG("Couldn't␣initialize␣semaphore");
    return -1;
  }

  /*****************************/
  /* Create ftt_dispatcher task */
  /*****************************/
  rtl_pthread_attr_init( &attr );
  sched_param.sched_priority = rtl_sched_get_priority_max(RTL_SCHED_FIFO);
  rtl_pthread_attr_setschedparam( &attr, &sched_param );
#ifdef RTL_PRO
  rtl_pthread_attr_setfp_np(&attr, 1);
#endif
  if ( (ret=rtl_pthread_create( &ftt_dispatcher_th, &attr, ftt_dispatcher,
      (void *)0 )) ){
    ERROR_MSG("Init␣Fatal␣Error:␣pthread_create␣returned␣%d", ret);
    return -2;
  }
#ifdef PRINT_THREAD_ID
#ifdef RTL_PRO
  ERROR_MSG("New␣thread␣id:␣%d", ftt_dispatcher_th);
```

```
#else
  ERROR_MSG("New␣thread␣id:␣%lu", ftt_dispatcher_th);
#endif
#endif

#ifdef PRINT_MODULES_INIT
  PRINT_MSG("␣-M_DispatcherInit␣Ok");
#endif
  return 0;
}

void M_DispatcherClose(void)
{
  /* Clean up the semaphore */
  FTT_SEMAPHORE_destroy(&start_dispatcher);

  /* cancel the ftt_dispatcher thread */
  rtl_pthread_cancel( ftt_dispatcher_th );
  rtl_pthread_join( ftt_dispatcher_th, NULL );

#ifdef PRINT_MODULES_INIT
  PRINT_MSG("␣-M_DispatcherClose␣Ok");
#endif
}

void M_DispatcherStartSystem(void)
{
  FTT_SEMAPHORE_post(&start_dispatcher);
  return ;
}
```

A.2 Slave eth_filter

```
/*****************************************************************************
 * S_eth_filter.c:
 *****************************************************************************
 * Copyright (C) 2006-2012 the FTT-SE team.
 *
 * Author: Ricardo Marau <marau at fe.up.pt>
 *
 * FTT-SE is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  FTT-SE is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with FTT-SE.  If not, see <http://www.gnu.org/licenses/>.
 *****************************************************************************/

#include "Ports.h"
#include "ftt-global.h"

#include "ethernet.layer.h"

#include "S_ECscheduled.h"
#include "S_Dispatch.h"
#include "S_nodeID.h"
#include "S_db.h"

//#define SLAVE_TM_DECODE /* cfg define */
//#define SLAVE_WATCH_DOG /* cfg define */

#ifndef FTT_SLAVE_COMPILE
#undef SLAVE_WATCH_DOG
#endif


#if defined(L26) & defined(SLAVE_WATCH_DOG)
#include <unistd.h>
#include <signal.h>
#endif

extern void S_NRDB_enter(void);
extern void S_NRDB_leave(void);
```

```
static rtl_pthread_t synchronous_window_start_th;
struct rtl_timespec synchronous_window_activation;

static FTT_SEMAPHORE time_to_turn_around_semaph;
static FTT_SEMAPHORE dispatch_is_awake_semaph;
static char dispatch_is_awake;
struct rtl_timespec start;

/* Fault injection */
static unsigned char combination_index;
static unsigned char trigger_message_index;
static unsigned char fault_simulation;
static unsigned char num_errors;
static unsigned char combination_counter;
static unsigned char total_combinations;
struct rtl_errors faults;
```

```
#ifdef SLAVE_WATCH_DOG
#define SLAVE_WATCH_DOG_START_VALUE 20 /*20=19ECs*/
static unsigned long watch_ec_len=50; //ms
static int watch_dog_counter=0;
static rtl_pthread_t watch_dog_th;

static void *watch_dog(void *t)
{
  struct rtl_timespec next_activation; //used to time activations

  rtl_clock_gettime( RTL_CLOCK_REALTIME, &next_activation );
  while (1) {
    rtl_timespec_add_ns( &next_activation, watch_ec_len*1000000 );
    rtl_clock_nanosleep( RTL_CLOCK_REALTIME, RTL_TIMER_ABSTIME,
        &next_activation, NULL);
    if (watch_dog_counter)
      watch_dog_counter--;
    if (watch_dog_counter == 1){
#ifdef L26
      PRINT_MSG("Watch␣dog␣detected␣on␣my␣pid:%d", getpid() );
      kill( getpid(),SIGINT);
#else
      PRINT_MSG("Watch␣dog␣detected.␣Doing␣nothing␣for␣now." );
#endif
    }


  }
  return NULL;
}

#endif
```

```
static void *synchronous_window_start(void *t)
{
  struct rtl_timespec temp;
  unsigned char diff = 1;

  while ( diff ) {
    rtl_clock_gettime( RTL_CLOCK_REALTIME, &temp );

    /** CS **/
    FTT_SEMAPHORE_wait( &time_to_turn_around_semaph );

    if ( rtl_timespec_gt( temp, synchronous_window_activation ) ){
      diff = 0;
    }

    FTT_SEMAPHORE_post( &time_to_turn_around_semaph );
    /** CS **/


  }

  rtl_clock_gettime( RTL_CLOCK_REALTIME, &temp );

  if ( !dispatch_is_awake ){

    /** CS **/
    FTT_SEMAPHORE_wait( &dispatch_is_awake_semaph );

    dispatch_is_awake = 1;
    S_Dispatch_Wake();

    FTT_SEMAPHORE_post( &dispatch_is_awake_semaph );
    /** CS **/
```

```
#ifdef FTT_SLAVE_COMPILE
    printf("Turn␣Around␣window␣activation:␣%llu\n", rtl_timespec_getns(temp));
#endif
  }

  return NULL;
}
```

```
void S_eth_filter(unsigned char *rmsg, unsigned short size, struct rtl_timespec
    rx_timestamp)
{

  FTT_PKT_GENERIC_HEADERS *ftt_rmsg_pointer= (FTT_PKT_GENERIC_HEADERS *)rmsg;
  int i; /* General counters */

  //PRINT_MSG("S_eth_filter in");
  //PRINT_MSG("Slave_ethernet_pkt_rx called at %ld",
      rx_timestamp.tv_sec*1000000 + rx_timestamp.tv_nsec/1000 );

  if ( ntohs( ftt_rmsg_pointer->eth_header.eth_type ) != ETH_FTT_TYPE ) {
    //ERROR_MSG("Rx_event Warnning: Frame is not ETH_FTT_TYPE ");
    return;
  }

  /* Get ftt frame type */
  switch( ntohs(ftt_rmsg_pointer->pkt_header.type) )
  {
    case FTT_MST_MSG: /* TM received */
      {

        FTT_PKT_TM_HEADERS *ftt_tm_pkt = (FTT_PKT_TM_HEADERS *)ftt_rmsg_pointer;
        TM_SYNC_MSG_INDEX *tm_sm_element;
        TM_ASYNC_MSG_INDEX *tm_am_element;
        TM_SIG_ACK_INDEX *tm_sig_ack_element;

        unsigned short nsm,nam,nsa;
        static unsigned char EC_last_seq_no=255;
        static unsigned char first_run = 1;
```

```
        unsigned char num_tms_per_ec;
        unsigned char tm_seq_no;
        unsigned short tm_interarrival_time;

        unsigned short time_to_synchronous_window;
        unsigned short time_from_EC_start;


        tm_seq_no = ftt_tm_pkt->tm_reliable_header.tm_sequence_number;
        tm_interarrival_time =
            ftt_tm_pkt->tm_reliable_header.tm_interarrival_time;
        num_tms_per_ec = ftt_tm_pkt->tm_reliable_header.num_tms_per_ec;
```

```
        /* Fault injection simulation */

        if ( fault_simulation  ) {
          if ( !total_combinations ){
            int i;
            for (i = 0; i < num_tms_per_ec; i++ ){
              total_combinations += number_combinations( num_tms_per_ec, i );
            }
          }
          if ( !combination_index && !trigger_message_index ){
            /* Return 0 in the array position corresponding to the TM that must
                fail */
```

```
            combinations ( num_tms_per_ec, num_errors, &faults );
            int i;
            int j;
            for ( i = 0; i<faults.num_combinations; i++ ){
              printf("Combinaci??n␣%d␣es␣:␣(␣", i);
              for ( j = 0; j < num_tms_per_ec; j++){
                printf("%d,␣", faults.combinations[i][j]);
              }
              printf(")\n");
            }
          }
        }

        /* Fault injection simulation */
```

```
        if ( !first_run ){
          if ( abs ( EC_last_seq_no - ftt_tm_pkt->tm_header.seq_no ) > 1 &&
            ( EC_last_seq_no != 255 || ftt_tm_pkt->tm_header.seq_no != 0 )){
            printf("TM:␣rx␣bad␣seq_no.␣Expected␣%d␣but␣received␣%d␣\n",
                EC_last_seq_no, ftt_tm_pkt->tm_header.seq_no);
          }
        }
        first_run = 0;


        if ( EC_last_seq_no != ftt_tm_pkt->tm_header.seq_no && (
            !fault_simulation ||
          ( fault_simulation &&
              faults.combinations[combination_index][tm_seq_no] ) ) ) {

          rtl_pthread_join( synchronous_window_start_th, NULL );
          dispatch_is_awake = 0;

        }

        if ( (fault_simulation &&
            faults.combinations[combination_index][tm_seq_no])
            || !fault_simulation ){

#ifdef FTT_SLAVE_COMPILE
          printf("TM:␣%d␣arrival:␣%llu\n", tm_seq_no,
              rtl_timespec_getns(rx_timestamp));
#endif
        }

        /** CS **/
        FTT_SEMAPHORE_wait( &dispatch_is_awake_semaph );

        if ( !dispatch_is_awake && ( !fault_simulation ||
          ( fault_simulation &&
              faults.combinations[combination_index][tm_seq_no] ) ) ){
```

```
          if ( S_Dispatcher_is_running() ){
            printf("Analysing␣a␣new␣TM␣while␣S_Dispatcher␣still␣running\n");
            /* This means that the dispatcher is taking too much time to encode
                and notify Tx */
                    while ( S_Dispatcher_is_running() ) PRINTF("=");
          }


//TODO get the TM size and post the time of TM transmission to avoid jitter
```

```
          /* Time to Turn Around window */
          time_to_synchronous_window = (num_tms_per_ec -
              (tm_seq_no+1))*tm_interarrival_time;
```

```
        /** CS **/
        FTT_SEMAPHORE_wait( &time_to_turn_around_semaph );

        /* Update the synchronous window start moment */
        synchronous_window_activation = rx_timestamp;
        rtl_timespec_add_ns( &synchronous_window_activation,
            time_to_synchronous_window*1000 );

        FTT_SEMAPHORE_post( &time_to_turn_around_semaph );
        /** CS **/

        S_ECScheduled_StoreTimestamp( &synchronous_window_activation );

        if ( EC_last_seq_no != ftt_tm_pkt->tm_header.seq_no  ){

          S_Set_Master_mac ( ftt_tm_pkt->eth_header.eth_src );

          /* Let us store the TM contents in the internal EC table */
          S_ECScheduled_CleanTables();


          /* Get number of sync. and asynch. messages in the EC */
          nsm = ntohs(ftt_tm_pkt->tm_header.nsm);
          nam = ntohs(ftt_tm_pkt->tm_header.nam);
          nsa = ntohs(ftt_tm_pkt->tm_header.nsa);

          S_Dispatch_Set_ec_time( ftt_tm_pkt->tm_header.ec_time_ms );
#ifdef SLAVE_WATCH_DOG
          watch_ec_len = ftt_tm_pkt->tm_header.ec_time_ms;
#endif

#ifdef SLAVE_TM_DECODE
          PRINT_MSG("TM:");
#endif

          /* Store messages to produce in ECSchedule table */
          /* 1: First synchronous messages (and asynchronous via aperiodic
              server) */

          tm_sm_element = (TM_SYNC_MSG_INDEX *)(ftt_tm_pkt->data);
          tm_am_element = (TM_ASYNC_MSG_INDEX *)( ftt_tm_pkt->data +
              nsm*sizeof(TM_SYNC_MSG_INDEX) );
          tm_sig_ack_element = (TM_SIG_ACK_INDEX *)( ftt_tm_pkt->data +
              nsm*sizeof(TM_SYNC_MSG_INDEX) + nam*sizeof(TM_ASYNC_MSG_INDEX)
              );

          for(i=(nsm-1); i>=0; i--){
            /* add to table in reverse sort */
            S_ECScheduled_PutSMessage( ntohs(tm_sm_element[i].SMesgId),
                tm_sm_element[i].source_nodeID,
                ntohs(tm_sm_element[i].SMesg_frag_no) );
#ifdef SLAVE_TM_DECODE
            PRINTF("␣S%x-%d", ntohs(tm_sm_element[i].SMesgId),
                ntohs(tm_sm_element[i].SMesg_frag_no) );
#endif
          }

          for(i=(nam-1); i>=0; i--){
            S_ECScheduled_PutAMessage( ntohs (tm_am_element[i].AMesgId),
                tm_am_element[i].source_nodeID,
                ntohs(tm_am_element[i].AMesg_frag_no) );
#ifdef SLAVE_TM_DECODE
            PRINTF("␣A%x", ntohs(tm_am_element[i].AMesgId));
#endif
          }
```

```
            for(i=(nsa-1); i>=0; i--){
              if ( ntohs (tm_sig_ack_element[i].nodeID) == S_nodeID_read() ){
                PRINT_MSG ("Have to re-transmit seq_no: %d ",
                    tm_sig_ack_element[i].sig_seqno );
                S_ECScheduled_PutSigRequest( ntohs
                    (tm_sig_ack_element[i].nodeID),
                    tm_sig_ack_element[i].sig_seqno );
              }
            }

          }


          /* Activate the task that will send the messages (at appropriate time
              instants) to the ETh bus */
//          NDB_printmesg();
```

```
          /* Only create the thread if it's the first TM received in the EC */
          if ( EC_last_seq_no != ftt_tm_pkt->tm_header.seq_no ){
            if ( rtl_pthread_create( &synchronous_window_start_th, NULL,
                synchronous_window_start , NULL ) < 0 ){
              ERROR_MSG("Init Fatal Error: pthread_create");
              return;
            }
          }

          EC_last_seq_no = ftt_tm_pkt->tm_header.seq_no;
```

```
#ifdef SLAVE_WATCH_DOG
          watch_dog_counter = SLAVE_WATCH_DOG_START_VALUE;
#endif

        }
```

```
        FTT_SEMAPHORE_post( &dispatch_is_awake_semaph );
        /** CS **/
```

```
        /* Fault injection simulation */

        if  ( fault_simulation ){
          trigger_message_index++;
          if ( trigger_message_index == num_tms_per_ec ){
            trigger_message_index = 0;

            /*Esclavo 1*/

            if ( S_nodeID_read() == 1 ){
              combination_counter++;
              if ( combination_counter == total_combinations ){
                combination_counter = 0;
                combination_index++;
                if ( combination_index == faults.num_combinations ){
                  combination_index = 0;
                  num_errors++;
                  if ( num_errors == num_tms_per_ec ){
                    exit(0);
                  }
                }
              }
```

```
            /*Esclavo 2*/

            } else if ( S_nodeID_read() == 2 ){
              combination_index++;
              if ( combination_index == faults.num_combinations ){
                combination_index = 0;
                num_errors++;
                if ( num_errors == num_tms_per_ec ){
                  num_errors = 0;
                  combination_counter++;
                  if ( combination_counter == total_combinations ){
                    exit(0);
                  }
                }
              }
            }
          }
        }

        /* Fault injection simulation */
      }
```

```
      break;

    case FTT_SDATA_MSG: /* Synchronous Data Message received */
      {
        FTT_PKT_SDM_HEADERS *ftt_sdm_pointer = (FTT_PKT_SDM_HEADERS
            *)ftt_rmsg_pointer;

//        PRINT_MSG("Received SDATA_MESG %x  %d", ntohs(
    ftt_sdm_pointer->sdm_header.id ), ntohs(
    ftt_sdm_pointer->sdm_header.frag_seqno ) );
//PRINTF("s");

        if( NDB_test_consume_and_bind( ntohs( ftt_sdm_pointer->sdm_header.id )
            ) < 0 ) {
          //PRINT_MSG("This SM is not a consumer or not bound of this...");
          break;
        }
        //PRINT_MSG("Is a consumer of this SM");

        S_NRDB_enter();
        NDB_set_data2rx(
            ntohs( ftt_sdm_pointer->sdm_header.id ),
            ftt_sdm_pointer->sdm_header.source_nodeID,
            ntohs( ftt_sdm_pointer->sdm_header.frag_seqno ),
            ftt_sdm_pointer->data,
            (ftt_sdm_pointer->sdm_header.flag_s & TXNEWDAT_MASK),
#ifdef QNX_TAGGING
            ntohi( ftt_sdm_pointer->sdm_header.tagging ),
#endif
            rx_timestamp);
        S_NRDB_leave();
      }
      break;

    case FTT_ADATA_MSG: /* Asynchronous Data Message received */
      {
        FTT_PKT_ADM_HEADERS *ftt_adm_pointer = (FTT_PKT_ADM_HEADERS
            *)ftt_rmsg_pointer;

//        PRINT_MSG("Received ADATA_MESG %x", ntohs(
    ftt_adm_pointer->adm_header.id ));
```

```
        if( NDB_test_consume_and_bind( ntohs( ftt_adm_pointer->adm_header.id )
            ) < 0 ) {
        //  PRINTF(" will NOT consume or not ready to consume");
          break;
        }
        //PRINTF(" will consume");

        S_NRDB_enter();
        NDB_set_data2rx(
            ntohs( ftt_adm_pointer->adm_header.id ),
            ftt_adm_pointer->adm_header.source_nodeID,
            ntohs( ftt_adm_pointer->adm_header.frag_seqno),
            ftt_adm_pointer->data,
            (ftt_adm_pointer->adm_header.flag_s & TXNEWDAT_MASK),
#ifdef QNX_TAGGING
            ntohi( ftt_adm_pointer->adm_header.tagging ),
#endif
            rx_timestamp);
        S_NRDB_leave();
      }
      break;

    case FTT_PLUGnPLAY:
    case FTT_ASTATUS_MSG: /* Asynchronous Data Message received */
      //PRINT_MSG("Received ASTATUS_MESG_ID");
      break;

    case FTT_IDLE:
      {
        FTT_PKT_IDLE_HEADERS *ftt_idle_pointer = (FTT_PKT_IDLE_HEADERS
            *)ftt_rmsg_pointer;
        PRINT_MSG("IDDLE␣time␣AM␣message:␣%x", ntohs(
            ftt_idle_pointer->idle_header.id ) );
      }
      break;

    default:
      ERROR_MSG("␣Received␣an␣ilegal␣message␣id");
      break;

  } /* Switch(frametype) */
  return ;
}

signed char S_eth_filter_Init(void)
{
```

```
  /* Fault simulation */
  fault_simulation = 1;
  combination_counter = 0;
  total_combinations = 0;
  combination_index = 0;
  trigger_message_index = 0;
  num_errors = 0;
  /* Init semaphores */
  if ( FTT_SEMAPHORE_init( &time_to_turn_around_semaph, 1 ) < 0){
      //Initialization locked
    ERROR_MSG("Couldn't␣initialize␣semaphore");
    return -1;
  }
  if ( FTT_SEMAPHORE_init( &dispatch_is_awake_semaph, 1 ) < 0){
      //Initialization locked
    ERROR_MSG("Couldn't␣initialize␣semaphore");
    return -1;
  }
```

```
#ifdef FTT_SLAVE_COMPILE
  PRINT_MSG("Initializing␣the␣S_eth_filter␣as␣the␣main␣receiver");
  ETH_L_rx_fun_register( S_eth_filter );

#ifdef SLAVE_WATCH_DOG
  if ( rtl_pthread_create( &watch_dog_th, NULL, watch_dog, NULL )<0 ){
    ERROR_MSG("Init␣Fatal␣Error:␣pthread_create");
    return -1;
  }
#endif

#else //FTT_MASTER_COMPILE

#endif

#ifdef PRINT_MODULES_INIT
  PRINT_MSG("␣-S_eth_filter_Init␣Ok");
#endif

  return 0;

}

void S_eth_filter_Close(void)
{

  FTT_SEMAPHORE_destroy( &time_to_turn_around_semaph );
  FTT_SEMAPHORE_destroy( &dispatch_is_awake_semaph );

#ifdef FTT_SLAVE_COMPILE
#ifdef SLAVE_WATCH_DOG
  rtl_pthread_cancel( watch_dog_th );
  rtl_pthread_join( watch_dog_th, NULL );
#endif
  ETH_L_rx_fun_unregister();
#endif

#ifdef PRINT_MODULES_INIT
  PRINT_MSG("␣-S_eth_filter_Close␣Ok");
#endif
}
```

A.3 Slave dispatcher

```
/*******************************************************************************
 * S_Dispatch.c:
 *******************************************************************************
 * Copyright (C) 2006-2012 the FTT-SE team.
 *
 * Author: Ricardo Marau <marau at fe.up.pt>
 *
 * FTT-SE is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  FTT-SE is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with FTT-SE.  If not, see <http://www.gnu.org/licenses/>.
 *******************************************************************************/

#include "Ports.h"

#include "ftt-global.h"
#include "ethernet.layer.h"
#include "S_ECscheduled.h"
#include "S_db.h"
#include "S_nodeID.h"
#include "S_igmp_queue.h"
#include "S_appPnP.h"
#include "S_sig_backup.h"

#include <unistd.h>


//#define DISABLE_DISPATCHING
//#define DISPATCHING_DECODE
//#define BE_POLITE
//#define DISABLE_AM_SM_TX
//#define DISABLE_ASM_TX
//#define DISABLE_TURNAROUND_WAIT
#define DISABLE_SIGNALING_SYNCH_DELAY /*removed because of some signaling
    messages were being lost... in L26*/



/*!
\brief Process ID of scheduler task
*/
static rtl_pthread_t ftt_sdispatcher_th;
static FTT_SEMAPHORE sdisp_wake_semaph;

static unsigned int S_ec_period=0; //us

void *ftt_sdispatcher(void *t);

signed char S_Dispatch_Init( void )
{
  rtl_pthread_attr_t attr;
  struct rtl_sched_param sched_param;

  int ret;
```

```
  /* Init semaphore to wake the sDispatcher */
  if ( FTT_SEMAPHORE_init(&sdisp_wake_semaph, 0) < 0){  //Initialization locked
    ERROR_MSG("Couldn't␣initialize␣semaphore");
    return -1;
  }

  /* Create ftt_sDispatcher task */
  rtl_pthread_attr_init( &attr );
  sched_param.sched_priority = rtl_sched_get_priority_max(RTL_SCHED_FIFO)-1;
  rtl_pthread_attr_setschedparam( &attr, &sched_param );
#ifdef RTL_PRO
  rtl_pthread_attr_setfp_np(&attr, 1);
#endif
//  rtl_pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  if ( (ret = rtl_pthread_create( &ftt_sdispatcher_th, &attr, ftt_sdispatcher,
       (void *)0 ))<0 ){
    ERROR_MSG("Init␣Fatal␣Error:␣pthread_create␣returned␣%d", ret);
    return -2;
  }
#ifdef PRINT_THREAD_ID
#ifdef RTL_PRO
  ERROR_MSG("New␣thread␣id:␣%d", ftt_sdispatcher_th);
#else
  ERROR_MSG("New␣thread␣id:␣%lu", ftt_sdispatcher_th);
#endif
#endif

#ifdef PRINT_MODULES_INIT
  PRINT_MSG("␣-S_Dispatch_Init␣Ok");
#endif
  return 0;
}

void S_Dispatch_Close(void)
{

//  PRINT_MSG("Closing S_Dispatch");
  /* cancel the ftt_dispatcher thread */
  rtl_pthread_cancel( ftt_sdispatcher_th );
  rtl_pthread_join( ftt_sdispatcher_th, NULL );

  /* Clean up the semaphore */
  FTT_SEMAPHORE_destroy(&sdisp_wake_semaph);

#ifdef PRINT_MODULES_INIT
  PRINT_MSG("␣-S_Dispatch_Close␣Ok");
#endif

}

void S_Dispatch_Wake(void)
{
  /* Activate ftt_sdispatcher() */
  FTT_SEMAPHORE_post(&sdisp_wake_semaph);
}

void S_Dispatch_Set_ec_time( int ec_time_ms )
{
  S_ec_period = ec_time_ms*1000;
  return;
}

unsigned int S_Dispatch_Read_ec_time_us( void )
{
  return S_ec_period;
}
```

```
static unsigned char S_Dispatcher_running_flag;

unsigned char S_Dispatcher_is_running(void)
{
  return (S_Dispatcher_running_flag != 0);
}

extern void S_NRDB_enter(void);
extern void S_NRDB_leave(void);


void *ftt_sdispatcher(void *t)
/*****************************************************************************/
/*!
\brief This task is responsible by timely produce the station messages.

  This task is responsible by timely produce the station messages.
    When an Ethernet frame with a Trigger message arrives, the routine\n
  that handles the Ethernet frame processing fills a data structure\n
  with the IDs and release time of the messages to be produced by this
      station.\n
  When the Trig mesg is completely processed, it activates this task.

*****************************************************************************/
{
  int ret;

  unsigned short MesgID;    /* Message ID */
  unsigned short Mesg_frag_no;    /* Message Length (in slots) */
  unsigned char  Mesg_source_nodeID;
  signed char mesgstatus;   /* Status ofa given message */

  unsigned char *tmsg;
  FTT_PKT_GENERIC_HEADERS *ftt_tmsg_generic_pointer;
  FTT_PKT_SDM_HEADERS *ftt_tmsg_sdm_pointer;
  FTT_PKT_ADM_HEADERS *ftt_tmsg_adm_pointer;

  struct rtl_timespec jitter;

  unsigned short mesg_efflen;    /* Real data length of message (in bytes) */
  unsigned char mesg_flags=0;
  unsigned char mesg_seqn;        /* Message sequence number */
  unsigned short frag_seqn; /* Message fragmentation seqno*/

  FTT_SIGNAL *callback_sig;

  unsigned char *mac=NULL;

#ifdef QNX_TAGGING
  QNX_TAGGING_TYPE tagging;
#endif

  while (1){
    S_Dispatcher_running_flag = 0;
    /* Wait for start */
    if ( FTT_SEMAPHORE_wait(&sdisp_wake_semaph) < 0 )
      return NULL;

#ifndef DISABLE_TURNAROUND_WAIT
    /* 2 - Waits for time-to-txmit */
    /* 3 - Busy waits the arrival of correct instant */
    {
```

```
      struct timespec tm_rx_abs_time;

      /* Get the TM absolute time reference */
      tm_rx_abs_time = S_ECScheduled_ReadTimestamp();
      timespec_add_ns( &tm_rx_abs_time, TURN_AROUND_WINDOW * 1000);

      struct rtl_timespec temp;
      rtl_clock_gettime( RTL_CLOCK_REALTIME, &temp );
      while ( rtl_timespec_gt( tm_rx_abs_time, temp ) ){
        rtl_clock_gettime( RTL_CLOCK_REALTIME, &temp );
      }
#ifdef FTT_SLAVE_COMPILE
      printf("Dispatch␣wake␣activation:␣%llu␣\n", rtl_timespec_getns(temp));
#endif
```

```
    }
#endif

    S_Dispatcher_running_flag = 1;
#ifdef DISABLE_DISPATCHING
    continue;
#endif

#if defined(BE_POLITE)||defined(DISPATCHING_DECODE)
    PRINT_MSG("S_Dispatcher:␣");
#endif

    /* Lets go through the table */
    while ( S_ECScheduled_GetMessage(&MesgID, &Mesg_source_nodeID,
        &Mesg_frag_no) ){

#ifdef DISPATCHING_DECODE
    PRINT_MSG("␣␣Var:%x␣frag%d␣src:%d", MesgID, Mesg_frag_no,
        Mesg_source_nodeID );
#endif

      if ( (Mesg_source_nodeID != S_nodeID_read()) ||
           ((mesgstatus = NDB_test_produce_and_bind(MesgID, &mac)) < 0) ){
        /* Station not producer of this message*/
#ifdef DISPATCHING_DECODE
        PRINTF("␣pass");
#endif
        continue ; /* and nothing else matters...Metallica*/
      }
#ifdef DISPATCHING_DECODE
      PRINTF("␣tx_ing");
#endif

      /* Get one new tx buffer */
      if ( ETH_L_tx_reserve_buffer( &tmsg ) < 0 ){
        ERROR_MSG("Problems␣getting␣a␣new␣Tx␣buffer␣-␣Progamming␣error␣");
      }

      /* 1 - Prepare the message */
      ftt_tmsg_generic_pointer = (FTT_PKT_GENERIC_HEADERS *)tmsg;

      /* The source and destination address */
      ETH_L_CopyMAC_to_from( ftt_tmsg_generic_pointer->eth_header.eth_dest, mac
          );
      ETH_L_CopyMAC_to_from( ftt_tmsg_generic_pointer->eth_header.eth_src,
          S_Get_My_mac() );

      /* Message type */
      ftt_tmsg_generic_pointer->eth_header.eth_type = htons ( ETH_FTT_TYPE );
```

```
      mesg_efflen = 0;
      callback_sig = NULL;

      /* Select message type */
      if( FTT_VAR_ID_IS_ID_SYNC(MesgID) ){
        //PRINT_MSG("SDidp: SDATA_MESG_ID");
        ftt_tmsg_sdm_pointer = (FTT_PKT_SDM_HEADERS *)(tmsg);

        /* Get data into net buffer */
        S_NRDB_enter();
        mesgstatus = NDB_get_data2tx(
            MesgID,
            Mesg_frag_no,
            ftt_tmsg_sdm_pointer->data,
            &mesg_efflen,
            &mesg_seqn,
            &frag_seqn,
#ifdef QNX_TAGGING
            &tagging,
#endif
            &callback_sig );
        S_NRDB_leave();

        //PRINT_MSG(" NDB_get_data2tx ret: %d", mesgstatus);
//if (frag_seqn == 1) PRINTF(".");
//PRINTF("S");
        mesg_efflen = FTT_PKT_SDM_LOGICALSIZE_B( mesg_efflen );

        if(mesgstatus < 0) {
          ERROR_MSG("Strange␣Error:␣ftt_get2tx_sdata␣failed␣with␣0x%x␣frag:%d␣
              ", MesgID, Mesg_frag_no );
          ETH_L_tx_un_reserve_buffer( );
          continue;
        }

        mesg_flags = 0;
        if(mesgstatus)   /* Update the TXNEWDAT bit on the message identifier */
          mesg_flags |= TXNEWDAT_MASK;

        /* Setup message ID */
        ftt_tmsg_sdm_pointer->pkt_header.type = htons( FTT_SDATA_MSG );
        ftt_tmsg_sdm_pointer->sdm_header.id = htons((unsigned short)MesgID);
        ftt_tmsg_sdm_pointer->sdm_header.seq_no = mesg_seqn;
        ftt_tmsg_sdm_pointer->sdm_header.flag_s = mesg_flags;
        ftt_tmsg_sdm_pointer->sdm_header.frag_seqno = htons(frag_seqn);
        ftt_tmsg_sdm_pointer->sdm_header.source_nodeID = S_nodeID_read();
#ifdef QNX_TAGGING
        ftt_tmsg_sdm_pointer->sdm_header.tagging  = htoni(tagging);
#endif

      }
      else {
        //PRINT_MSG("SDidp: ADATA_MESG_ID");
        ftt_tmsg_adm_pointer = (FTT_PKT_ADM_HEADERS *)(tmsg);

        /* Get data into net buffer */
        S_NRDB_enter();
        mesgstatus=NDB_get_data2tx(
            MesgID,
            Mesg_frag_no,
            ftt_tmsg_adm_pointer->data,
            &mesg_efflen,
            &mesg_seqn,
            &frag_seqn,
#ifdef QNX_TAGGING
            &tagging,
```

```
#endif
            &callback_sig );
          S_NRDB_leave();
          //PRINT_MSG(" ftt_get2tx_adata ret: %d", mesgstatus);

          /* If no message was in the FIFO */
          if (mesg_efflen == 0)
            ftt_tmsg_adm_pointer->pkt_header.type = htons( FTT_IDLE );
          else
            ftt_tmsg_adm_pointer->pkt_header.type = htons( FTT_ADATA_MSG );

          mesg_efflen = FTT_PKT_ADM_LOGICALSIZE_B( mesg_efflen );

          if(mesgstatus < 0) {
            ERROR_MSG("Strange␣Error:␣ftt_get2tx_adata␣failed␣after␣
                ftt_test_producer");
            ETH_L_tx_un_reserve_buffer( );
            continue;
          }

          mesg_flags = 0;
          if(mesgstatus)   /* Update the TXNEWDAT bit on the message identifier */
            mesg_flags |= TXNEWDAT_MASK;

//          PRINT_MSG("sdispatcher: Async mesg_eff_len = %d", mesg_efflen);

          /* Setup message ID */
          ftt_tmsg_adm_pointer->adm_header.id  = htons((unsigned short)MesgID);
          ftt_tmsg_adm_pointer->adm_header.seq_no = mesg_seqn;
          ftt_tmsg_adm_pointer->adm_header.flag_s  = mesg_flags;
          ftt_tmsg_adm_pointer->adm_header.frag_seqno  = htons(frag_seqn);
          ftt_tmsg_adm_pointer->adm_header.source_nodeID = S_nodeID_read();
#ifdef QNX_TAGGING
          ftt_tmsg_adm_pointer->adm_header.tagging  = htoni(tagging);
#endif

#ifndef DISABLE_TURNAROUND_WAIT
        /* 2 - Waits for time-to-txmit */
        /* 3 - Busy waits the arrival of correct instant */
        {
          struct timespec tm_rx_abs_time;

          /* Get the TM absolute time reference */
          tm_rx_abs_time = S_ECScheduled_ReadTimestamp();

          timespec_add_ns( &tm_rx_abs_time, SCAN_WINDOW * 1000);
          jitter = timespec_from_ns(0);
          rtl_clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME|RTL_TIMER_ADVANCE,
              &tm_rx_abs_time, &jitter);
        }
#endif
        }

#ifndef DISABLE_AM_SM_TX
        /* 4 - Send packet */
//        PRINT_MSG ("Dispatching a packet with size: %d", mesg_efflen );
        if ( (ret = ETH_L_tx_send_messagebuffer(NULL, mesg_efflen)) < 0 ){
          ERROR_MSG("Message␣not␣sent.␣ETH_L_tx_send_messagebuffer␣returned:␣%d",
              ret);
          /* Maybe I should un-reserv the buffer tmsg */
        }
#endif
        /* 5 - Activate callback (if defined) */
        if(callback_sig!=NULL) {
          FTT_signal( callback_sig );
        }
```

```
    } // while get message

    S_Dispatcher_running_flag = 0;

#ifdef FTT_SLAVE_COMPILE
  {
    ASM_MSG_QUEUE_INDEX *temp_p;
    unsigned short counter;
    signed short nodeID;
    FTT_PKT_ASM_HEADERS *ftt_tmsg_asm_pointer;
    unsigned char app_id;
    static unsigned char signaling_seq_no=0;

    NDB_AM_Get_sent_RETURN_TYPE ndb_return_AM_queues[STA_MAX_AM_VARS];
    unsigned short ndb_return_buf_size = STA_MAX_AM_VARS;


    if ( ETH_L_tx_reserve_buffer( &tmsg ) < 0 ){
      ERROR_MSG("Problems␣getting␣a␣new␣Tx␣buffer␣-␣Progamming␣error␣");
    }

    ftt_tmsg_asm_pointer = (FTT_PKT_ASM_HEADERS *)tmsg;

    switch (nodeID=S_nodeID_read()){
      case -1:
        ERROR_MSG("␣master_mac␣still␣not␣initialized");
        break;
      case -2:
        /* Send the Boot request */
//        PRINT_MSG("Preparing to send a PnP req");
        ETH_L_CopyMAC_to_from( ftt_tmsg_asm_pointer->eth_header.eth_dest,
            S_Get_Master_mac() );
        ETH_L_CopyMAC_to_from( ftt_tmsg_asm_pointer->eth_header.eth_src,
            S_Get_My_mac() );

        ftt_tmsg_asm_pointer->eth_header.eth_type = htons ( ETH_FTT_TYPE );
        ftt_tmsg_asm_pointer->pkt_header.type = htons( FTT_PLUGnPLAY );

        //Missing the Seq_no
        ftt_tmsg_asm_pointer->asm_header.seq_no = signaling_seq_no;
        ftt_tmsg_asm_pointer->asm_header.nodeID = (unsigned char)nodeID;
        ftt_tmsg_asm_pointer->asm_header.nam = htons((unsigned short)0);
        mesg_efflen = FTT_PKT_ASM_LOGICALSIZE_B_from_no(0);
        break;
      default:
//        PRINT_MSG("Preparing to send an ASM");
        /* Send the Asynch buffers status */
        ETH_L_CopyMAC_to_from( ftt_tmsg_asm_pointer->eth_header.eth_dest,
            S_Get_Master_mac() );
        ETH_L_CopyMAC_to_from( ftt_tmsg_asm_pointer->eth_header.eth_src,
            S_Get_My_mac() );

        ftt_tmsg_asm_pointer->eth_header.eth_type = htons ( ETH_FTT_TYPE );
        ftt_tmsg_asm_pointer->pkt_header.type = htons( FTT_ASTATUS_MSG );

        temp_p = (ASM_MSG_QUEUE_INDEX *)ftt_tmsg_asm_pointer->data;

        S_NRDB_enter ();
        NDB_AM_Get_sent( ndb_return_AM_queues, &ndb_return_buf_size);
        S_NRDB_leave();

        counter = 0;
        /* check if there is any app request */
        if (S_appPnP_is_there_a_request(&app_id)){
          // We will notify that will as if it was an assync notification
```

```
          // The AMesgId field will hold the MASTER_BROADCAST_A_CH_ID
          //   - (not used in this context - so used to notify this request)
          // The AMesgQueueLen holds the app_id
          temp_p[counter].AMesgId = htons(MASTER_BROADCAST_A_CH_ID);
          temp_p[counter].AMesgQueueLen = htons((unsigned short)app_id);
          counter++;
        }

        /* now, the real AM status */
        while ( ndb_return_buf_size-- ){
//          PRINT_MSG("Sending status for id: %x
    num:%d",ndb_return_AM_queues[ndb_return_buf_size].tagged_id,
    ndb_return_AM_queues[ndb_return_buf_size].num );
          temp_p[counter].AMesgId =
            htons(ndb_return_AM_queues[ndb_return_buf_size].tagged_id);
          temp_p[counter].AMesgQueueLen =
            htons(ndb_return_AM_queues[ndb_return_buf_size].num);
          counter++;
        }

#ifdef SIG_RECOVER
        {
        unsigned char seq_no_retransmit;
        void *ret_mem_p;
        unsigned short ret_mem_size;

        while ( S_ECScheduled_GetSigRequest( NULL, &seq_no_retransmit) ){
          S_sig_backup_pull( seq_no_retransmit, &ret_mem_p, &ret_mem_size );

          if ( ret_mem_size%sizeof(ASM_MSG_QUEUE_INDEX) ) {
            ERROR_MSG("Programming␣error!␣Size␣must␣always␣come␣as␣a␣multiple␣
                of␣sizeof(ASM_MSG_QUEUE_INDEX)␣");
          }
          PRINT_MSG("Recovering␣signaling␣%d.␣Caught␣%d␣IDXs",
              seq_no_retransmit, ret_mem_size/sizeof(ASM_MSG_QUEUE_INDEX) );

          if (ret_mem_p) {
            memcpy( (void *)(temp_p+counter), ret_mem_p, ret_mem_size );
            counter += ret_mem_size/sizeof(ASM_MSG_QUEUE_INDEX);
          }

        }

        S_sig_backup_push( signaling_seq_no, temp_p,
            counter*sizeof(ASM_MSG_QUEUE_INDEX) );
        }
#endif

        ftt_tmsg_asm_pointer->asm_header.seq_no = signaling_seq_no;
        ftt_tmsg_asm_pointer->asm_header.nodeID = (unsigned char)nodeID;
        ftt_tmsg_asm_pointer->asm_header.nam = htons((unsigned short)counter);
        mesg_efflen = FTT_PKT_ASM_LOGICALSIZE_B_from_no(counter);

        break;
    }
    signaling_seq_no ++;

#ifndef DISABLE_SIGNALING_SYNCH_DELAY
#ifndef ICNOVA
    {
      struct timespec tm_rx_abs_time;

      /* Get the TM absolute time reference */
      tm_rx_abs_time = S_ECScheduled_ReadTimestamp();
```

```
      //timespec_add_ns( &tm_rx_abs_time, ( S_ec_period-20 ) * 1000); //como
          estava antes
      timespec_add_ns( &tm_rx_abs_time, ( S_ec_period-(S_ec_period*0.15) ) *
          1000);

      jitter = timespec_from_ns(0);
      //PRINTF(">Waiting");
      rtl_clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME|RTL_TIMER_ADVANCE,
          &tm_rx_abs_time, NULL /*jitter*/);
    }
#endif
#endif

#ifndef DISABLE_ASM_TX
    if ( (ret = ETH_L_tx_send_messagebuffer(NULL, mesg_efflen)) < 0 ){
      ERROR_MSG("Message␣not␣sent.␣ETH_L_tx_send_messagebuffer␣returned:␣%d",
          ret);
      /* Perhaps I should un-reserv the buffer tmsg */
    }
//    PRINTF(">Sent");
#endif
  }
#endif

    /* Flush the igmp_queue */
    S_igmp_queue_flush_queue();

  }//while

  return NULL;
}
```

A.4 Common ftt-global

```
/*****************************************************************************
 * ftt-global.h:
 *****************************************************************************
 * Copyright (C) 2006-2012 the FTT-SE team.
 *
 * Author: Ricardo Marau <marau at fe.up.pt>
 *
 * FTT-SE is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  FTT-SE is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with FTT-SE.  If not, see <http://www.gnu.org/licenses/>.
 *****************************************************************************/

#ifndef  FTT_GLOBAL_H
#define  FTT_GLOBAL_H

//////////Linux compiling slowing factor
#define L26_SLOWING 1  // Using a factor smaller than 100*1ms is dangerous in
    L26
                       //remember that the L26 resolution is 10 ms


#define MST_MAX_SM_VARS 100
#define MST_MAX_AM_VARS 100

#define STA_MAX_SM_VARS 10
#define STA_MAX_AM_VARS 10  //influencia o tamanho sa status ackowledge poll


/** @brief Maximum # of messages in 1 EC (Synchronous and Asynchronous) */
#define EC_MAX_SM_VARS  50  /* isto influencia o tamanho maximo da TM*/
#define EC_MAX_AM_VARS  300
#define EC_MAX_VARS (EC_MAX_SM_VARS + EC_MAX_AM_VARS)
#define EC_MAX_SIG_MISSING_REQUESTS 50

/** @brief Maximum valid ID of messages */
//#define MAX_ID     0x07FF
#define MAX_ID     (MESGID_MASK)

/** @brief Maximum number of messages that a Station can handle */
#define STA_MAX_VAR  50

#ifdef SW_UNICAST
#define SW_MAX_PORTS  10
#endif

/** @brief Maximum load allowed due to the scheduler (in %) */
#define SCHED_LOAD_BOUND 80

/** @brief Size (in # of messages) of each buffer associated with asynch mesgs
    transmitted or recieved locally by the node */
#define STA_AM_BUFF_SIZE  1
#define STA_SM_BUFF_SIZE  3

/** @brief Worst case len  of TM: Header(ID+Flags+nmesgs sinc and asinc +
    ID/Txi of sync. mesgs + ID of asynch mesgs + Prop delay */
```

```
#define TM_LEN_bits ((long)( FTT_PKT_TM_REALSIZE_B_from_no( EC_MAX_SM_VARS,
    EC_MAX_AM_VARS, EC_MAX_SIG_MISSING_REQUESTS ) )*8+FTT_IFS_bits)

#ifdef NODE_ASYNC_REPLY
#define POLL_STATUS_MAX_PAYLOAD ( sizeof(FTT_ASM_HEADER) +
    STA_MAX_AM_VARS*sizeof(FTT_ASM_MSG_QUEUE_INDEX) )
#define POLL_STATUS_LEN_bits
    ((long)(FRAME_REALSIZE_B(POLL_STATUS_MAX_PAYLOAD))*8+FTT_IFS_bits)
#endif

/** @brief Bit time in us (for 100Mb/s Ethernet) */
//#define BIT_TIME_US   0.1
#define BIT_TIME_US   ((float)0.01)

/** @brief EC length in bits */
#define EC_LEN_bits(_ec)    (long)(_ec/BIT_TIME_US)

/** @brief Percentage of the EC reserved to the Synch Phase */
//#define SWIN_SHARE  0.10
//#define SWIN_SHARE  0.85
#define SWIN_SHARE  0.10

/** @brief Percentage of the EC reserved to the Asynch Phase */
#define AWIN_SHARE      (float)(1-SWIN_SHARE) /* Percentage of the EC reserved
    to the Asynch Phase */
                    /* Note : These values should no be used in the code! */
          /* AWIN_SHARE + SWIN_SHARE cant be 100%, since some space is */
                    /*        used by the trigger message (and other control
                        messages?)*/
                    /*  Use in the code EC_MAX_bits_SP/AP that have already
                        included the overheads */

/** @brief Size of the scanning window (in us) after Trig mesg.\n
This time is intended to allow that all stations decode the TM and set the Tx
    of the first data message
Human rights equality for everyone... */
#define SCAN_WINDOW 0 /* us */

#define TURN_AROUND_WINDOW 0 /*US*/

//#define SCAN_WINDOW_BITS (long)(SCAN_WINDOW/BIT_TIME_US) /* bits */


#define REAL_IDLE_TIME  200 /* us - Time between the the TM and the
    transmission of the Synchs */
#if (REAL_IDLE_TIME > SCAN_WINDOW)
  #define DEAD_TIME (REAL_IDLE_TIME)
#else
  #define DEAD_TIME (SCAN_WINDOW)
#endif

#define DEAD_TIME_BITS (long)(DEAD_TIME/BIT_TIME_US) /* bits */

#define EC_EFF_LEN_bits(__ec) (long)(EC_LEN_bits(__ec)-SCAN_WINDOW-TM_LEN_bits)

/** @brief EC Maximum Synchronous Phase duration measured in bits */
#define EC_MAX_bits_SP(___ec) (long)( (EC_EFF_LEN_bits(___ec))*SWIN_SHARE)

/** @brief EC Maximum asynchronous Phase duration measured in bits */
#define EC_MAX_bits_AP(____ec)  (long)( (EC_EFF_LEN_bits(____ec)) -
    EC_MAX_bits_SP(____ec) )

/** @brief EC Maximum Synchronous Phase duration (in microsecs) */
#define EC_MAX_us_SP(____ec)  (long)(EC_MAX_bits_SP(____ec)*BIT_TIME_US)

/** @brief EC Maximum Asynchronous Phase duration (in microsecs) */
```

```
#define EC_MAX_us_AP(_____ec)     (long)(EC_MAX_bits_AP(_____ec)*BIT_TIME_US)

#ifdef NODE_ASYNC_REPLY
#define EC_MAX_us_POLL  ((long)((DEAD_TIME_BITS + TM_LEN_bits)*BIT_TIME_US))
#endif




/** @defgroup Ethernet_related_constants
 * Ethernet related constants
 * @{
 */

#define ETH_MAC_PREAMBLE_B  8 /* bytes */
#define ETH_MAC_DEST_B    6 /* bytes */
#define ETH_MAC_SOURCE_B  6 /* bytes */
#define ETH_MAC_TYPE_B    2 /* bytes */
#define ETH_MINIMUM_PAYLOAD 46  /* bytes */
#define ETH_MAXIMUM_PAYLOAD 1500/* bytes */
#define ETH_CRC_B     4 /* bytes */
#define ETH_INTERFRAMEGAP_B 12  /* bytes */


#define ETH_DATALEN_MIN (ETH_MINIMUM_PAYLOAD)
#define ETH_DATALEN_MAX (ETH_MAXIMUM_PAYLOAD)


/** @brief Overhead of an Ethernet Frame:\n
 Dest. Addr (6) + Src. Addr (6) + Type (2)\n
 Note : from a logical point of view, Prelude and FCS not included */
#define ETH_LOGICAL_OVERHEAD  (ETH_MAC_DEST_B + ETH_MAC_SOURCE_B +
    ETH_MAC_TYPE_B)
#define ETH_OVERHEAD_BYTES    ( ETH_MAC_PREAMBLE_B + ETH_LOGICAL_OVERHEAD +
    ETH_CRC_B + ETH_INTERFRAMEGAP_B )
#define ETH_OVERHEAD_bits ( ETH_OVERHEAD_BYTES*8 )

#define FRAME_LOGICALSIZE_B(_a) ( (_a) + ETH_LOGICAL_OVERHEAD )
#define FRAME_REALSIZE_B(_a)  ( (_a) + ETH_OVERHEAD_BYTES )

#define ETH_FRAME_HIDDEN_OVERHEAD ( ETH_MAC_PREAMBLE_B + ETH_CRC_B +
    ETH_INTERFRAMEGAP_B )
/** @} */




/** @brief Ethernet Frame type ID */
#define ETH_FTT_TYPE  0x8FF0

/** @brief Worst-case gap that must be forced between short FTT Data frames,
    due to non-simultaneous\n
reception of the TM mesg in all nodes and software jitter (on reception of TM /
    txmit of data frame */
#define FTT_IFS_bits          500

/** @brief Minimum allowed size for a data frame in FTT Ethernet\n
(to support bus and software jitter still without collisions) */
#define FTT_MINLEN_bits ((ETH_OVERHEAD_BYTES + ETH_DATALEN_MIN)*8 +
    FTT_IFS_bits)

/** @brief Value of the Slot time used as a count unit for message tx time
    indication in the trig. mesg */
#define SLOTV_US    10
```

```
#ifdef QNX_TAGGING
typedef unsigned int QNX_TAGGING_TYPE;
#endif

/* FTT Generic */
typedef struct{
  unsigned char eth_dest[ETH_MAC_DEST_B];
  unsigned char eth_src[ETH_MAC_DEST_B];
  unsigned short eth_type;
}__attribute__ ((packed)) FTT_ETH_HEADER;

typedef enum{
  FTT_MST_MSG = 0,
  FTT_SDATA_MSG,
  FTT_ADATA_MSG,
  FTT_ASTATUS_MSG,
  FTT_IDLE,
  FTT_PLUGnPLAY,
  FTT_NOTUSED = 0xFFFF /* only to force the enum size to 2B */
}__attribute__ ((packed)) FTT_PKT_TYPE;

typedef struct{
  FTT_PKT_TYPE type;
}__attribute__ ((packed)) FTT_PKT_HEADER;


typedef struct{
  FTT_ETH_HEADER eth_header;
  FTT_PKT_HEADER pkt_header;
}__attribute__ ((packed)) FTT_PKT_GENERIC_HEADERS;

#define FTT_PKT_GENERIC_max_payload ( ETH_DATALEN_MAX - sizeof(FTT_PKT_HEADER) )
#define FTT_PKT_GENERIC_min_payload ( ETH_DATALEN_MIN - sizeof(FTT_PKT_HEADER) )

/* TM */
typedef struct{
  unsigned char num_tms_per_ec;
  unsigned char tm_sequence_number;
  unsigned short  tm_interarrival_time;
} __attribute__ ((packed)) FTT_PKT_TM_RELIABLE_HEADER;

typedef struct{
  unsigned char  seq_no;
  unsigned char  ec_time_ms;
  unsigned short nsm;        ///< Number of synch. mesgs
  unsigned short nam;        ///< Number of asynch. mesgs
  unsigned short nsa;        ///< Number of signaling !ack
} __attribute__ ((packed)) FTT_PKT_TM_HEADER;

#define TM_SEQN_MASK     0x00FF

typedef struct{
  FTT_ETH_HEADER eth_header;
  FTT_PKT_HEADER pkt_header;

  FTT_PKT_TM_RELIABLE_HEADER tm_reliable_header;

  FTT_PKT_TM_HEADER tm_header;
  unsigned char data[];
} __attribute__ ((packed)) FTT_PKT_TM_HEADERS;

typedef struct{
  unsigned short SMesgId;  ///< ID
  unsigned short SMesg_frag_no;
  unsigned char  source_nodeID;
```

```
} __attribute__ ((packed)) TM_SYNC_MSG_INDEX;

typedef struct{
  unsigned short AMesgId;  ///< ID
  unsigned short AMesg_frag_no;
  unsigned char  source_nodeID;
} __attribute__ ((packed)) TM_ASYNC_MSG_INDEX;

typedef struct{
  unsigned short nodeID;   ///< ID
  unsigned char sig_seqno;
} __attribute__ ((packed)) TM_SIG_ACK_INDEX;

#define FTT_PKT_TM_DATALEN_MAX ( FTT_PKT_GENERIC_max_payload -
    sizeof(FTT_PKT_TM_HEADER))
#define FTT_PKT_TM_DATALEN_MIN ( FTT_PKT_GENERIC_min_payload -
    sizeof(FTT_PKT_TM_HEADER))
#define FTT_PKT_TM_LOGICALSIZE_B_from_no( _no_SM, _no_AM, _no_Sig_ACK ) (
    _no_Sig_ACK*sizeof(TM_SIG_ACK_INDEX) + _no_SM*sizeof(TM_SYNC_MSG_INDEX) +
    _no_AM*sizeof(TM_ASYNC_MSG_INDEX) + sizeof(FTT_PKT_TM_HEADERS) )
#define FTT_PKT_TM_REALSIZE_B_from_no( _no_SM, _no_AM, _no_Sig_ACK ) (
    _no_Sig_ACK*sizeof(TM_SIG_ACK_INDEX) + _no_SM*sizeof(TM_SYNC_MSG_INDEX) +
    _no_AM*sizeof(TM_ASYNC_MSG_INDEX) + sizeof(FTT_PKT_TM_HEADERS) +
    ETH_FRAME_HIDDEN_OVERHEAD)

/** @brief Mask for the 9th bit of identifier (flags) */
#define TXNEWDAT_MASK   0x01

/* SDM */
typedef struct{
  unsigned short id;         ///< Frame ID
  unsigned char seq_no;
  unsigned char flag_s;
  unsigned short frag_seqno; ///< Sequence number within the fragmentation
      (could be char)
  unsigned char  source_nodeID;
#ifdef QNX_TAGGING
  QNX_TAGGING_TYPE tagging;
#endif
} __attribute__ ((packed)) FTT_PKT_SDM_HEADER;

typedef struct{
  FTT_ETH_HEADER eth_header;
  FTT_PKT_HEADER pkt_header;
  FTT_PKT_SDM_HEADER sdm_header;
  unsigned char data[];
} __attribute__ ((packed)) FTT_PKT_SDM_HEADERS;

#define FTT_PKT_SDM_DATALEN_MAX ( FTT_PKT_GENERIC_max_payload -
    sizeof(FTT_PKT_SDM_HEADER))
#define FTT_PKT_SDM_DATALEN_MIN ( FTT_PKT_GENERIC_min_payload -
    sizeof(FTT_PKT_SDM_HEADER))
#define FTT_PKT_SDM_LOGICALSIZE_B( _load ) ( _load +
    sizeof(FTT_PKT_SDM_HEADERS) )
#define FTT_PKT_SDM_REALSIZE_B( _load ) ( _load + sizeof(FTT_PKT_SDM_HEADERS) +
    ETH_FRAME_HIDDEN_OVERHEAD)
#define FTT_PKT_SDM_REALSIZE_B_reverse( _load ) ( _load -
    sizeof(FTT_PKT_SDM_HEADERS) - ETH_FRAME_HIDDEN_OVERHEAD)

/* ADM */
typedef struct{
  unsigned short id;         ///< Frame ID
  unsigned char seq_no;
  unsigned char flag_s;
  unsigned short frag_seqno;  ///< Sequence number within the fragmentation
  unsigned char  source_nodeID;
```

```
#ifdef QNX_TAGGING
  QNX_TAGGING_TYPE tagging;
#endif
} __attribute__ ((packed)) FTT_PKT_ADM_HEADER;

typedef struct{
  FTT_ETH_HEADER eth_header;
  FTT_PKT_HEADER pkt_header;
  FTT_PKT_ADM_HEADER adm_header;
  unsigned char data[];
} __attribute__ ((packed)) FTT_PKT_ADM_HEADERS;

#define FTT_PKT_ADM_DATALEN_MAX ( FTT_PKT_GENERIC_max_payload -
    sizeof(FTT_PKT_ADM_HEADER))
#define FTT_PKT_ADM_DATALEN_MIN ( FTT_PKT_GENERIC_min_payload -
    sizeof(FTT_PKT_ADM_HEADER))
#define FTT_PKT_ADM_LOGICALSIZE_B( _load ) ( _load +
    sizeof(FTT_PKT_ADM_HEADERS) )
#define FTT_PKT_ADM_REALSIZE_B( _load ) ( _load + sizeof(FTT_PKT_ADM_HEADERS) +
    ETH_FRAME_HIDDEN_OVERHEAD)
#define FTT_PKT_ADM_REALSIZE_B_reverse( _load ) ( _load -
    sizeof(FTT_PKT_ADM_HEADERS) - ETH_FRAME_HIDDEN_OVERHEAD)

/* ASM */
typedef struct{
  unsigned char seq_no;
  unsigned char nodeID;
  unsigned short nam;
} __attribute__ ((packed)) FTT_PKT_ASM_HEADER;

typedef struct{
  FTT_ETH_HEADER eth_header;
  FTT_PKT_HEADER pkt_header;
  FTT_PKT_ASM_HEADER asm_header;
  unsigned char data[];
} __attribute__ ((packed)) FTT_PKT_ASM_HEADERS;

typedef struct{
  unsigned short AMesgId;  ///< ID
  unsigned short  AMesgQueueLen; ///< Len of the queue in the asynchronous
      producer
} __attribute__ ((packed)) ASM_MSG_QUEUE_INDEX;


#define FTT_PKT_ASM_LOGICALSIZE_B_from_no( _no ) (
    _no*sizeof(ASM_MSG_QUEUE_INDEX) + sizeof(FTT_PKT_ASM_HEADERS) )
#define FTT_PKT_ASM_REALSIZE_B_from_no( _no ) ( _no*sizeof(ASM_MSG_QUEUE_INDEX)
    + sizeof(FTT_PKT_ASM_HEADERS) + ETH_FRAME_HIDDEN_OVERHEAD)

/* IDLE */
typedef struct{
  unsigned short id;          ///< Frame ID
} __attribute__ ((packed)) FTT_PKT_IDLE_HEADER;

typedef struct{
  FTT_ETH_HEADER eth_header;
  FTT_PKT_HEADER pkt_header;
  FTT_PKT_IDLE_HEADER idle_header;
} __attribute__ ((packed)) FTT_PKT_IDLE_HEADERS;


////////////////////////////////////////////////////////////////////////////////

#define FTT_SM (0)
#define FTT_AM (1)
```

```
typedef unsigned short FTT_VAR_ID;   /* 1-type 3-tag 12-ID */

#define FTT_VAR_ID_TYPE_MASK (0x8000)
#define FTT_VAR_ID_TAG_MASK  (0x7000)
#define FTT_VAR_ID_ID_MASK   (0xFFF)

#define FTT_VAR_ID_TYPE_SHIFT (15)
#define FTT_VAR_ID_TAG_SHIFT (12)
#define FTT_VAR_ID_ID_SHIFT (0)

#define FTT_VAR_ID_TAG_BASEMASK  (0x7)

#define FTT_VAR_ID_GET_ID( _id ) ( ( (FTT_VAR_ID)_id & FTT_VAR_ID_ID_MASK ) >>
    FTT_VAR_ID_ID_SHIFT )
#define FTT_VAR_ID_GET_TAG( _id ) ( ( (FTT_VAR_ID)_id & FTT_VAR_ID_TAG_MASK )
    >> FTT_VAR_ID_TAG_SHIFT )
#define FTT_VAR_ID_GET_TYPE( _id ) ( ( (FTT_VAR_ID)_id & FTT_VAR_ID_TYPE_MASK )
    >> FTT_VAR_ID_TYPE_SHIFT )

#define FTT_VAR_ID_SET_ID( _idout, _idin ) { _idout = (( ((FTT_VAR_ID)_idin) <<
    FTT_VAR_ID_ID_SHIFT) & FTT_VAR_ID_ID_MASK) | ( ((FTT_VAR_ID)_idout) &
    (~FTT_VAR_ID_ID_MASK) ); }
#define FTT_VAR_ID_SET_TAG( _idout, _idin ) { _idout = (( ((FTT_VAR_ID)_idin)
    << FTT_VAR_ID_TAG_SHIFT) & FTT_VAR_ID_TAG_MASK) | ( ((FTT_VAR_ID)_idout) &
    (~FTT_VAR_ID_TAG_MASK) ); }
#define FTT_VAR_ID_SET_TYPE( _idout, _idin ) { _idout = (( ((FTT_VAR_ID)_idin)
    << FTT_VAR_ID_TYPE_SHIFT) & FTT_VAR_ID_TYPE_MASK) | ( ((FTT_VAR_ID)_idout)
    & (~FTT_VAR_ID_TYPE_MASK) ); }

#define FTT_VAR_ID_SET_TYPE_SYNC( _id )    ( FTT_VAR_ID_SET_TYPE( _id, FTT_SM )
    )
#define FTT_VAR_ID_SET_TYPE_ASYNC( _id )    ( FTT_VAR_ID_SET_TYPE( _id, FTT_AM
    ) )

#define FTT_VAR_ID_IS_ID_SYNC( __id )       ( FTT_VAR_ID_GET_TYPE( __id ) ==
    FTT_SM )
#define FTT_VAR_ID_IS_ID_ASYNC( __id )       ( FTT_VAR_ID_GET_TYPE( __id ) ==
    FTT_AM )

#define FTT_VAR_ID_CMP_ID( _id1, _id2)          ( ((FTT_VAR_ID)_id1 &
    FTT_VAR_ID_ID_MASK) == ((FTT_VAR_ID)_id2 & FTT_VAR_ID_ID_MASK) )
#define FTT_VAR_ID_CMP_TAG( _id1, _id2)          ( ((FTT_VAR_ID)_id1 &
    FTT_VAR_ID_TAG_MASK) == ((FTT_VAR_ID)_id2 & FTT_VAR_ID_TAG_MASK) )
#define FTT_VAR_ID_CMP_TYPE( _id1, _id2)          ( ((FTT_VAR_ID)_id1 &
    FTT_VAR_ID_TYPE_MASK) == ((FTT_VAR_ID)_id2 & FTT_VAR_ID_TYPE_MASK) )

#define FTT_VAR_ID_CMP_ID_TYPE( _id1, _id2)       ( ((FTT_VAR_ID)_id1 &
    (FTT_VAR_ID_ID_MASK|FTT_VAR_ID_TYPE_MASK)) == ((FTT_VAR_ID)_id2 &
    (FTT_VAR_ID_ID_MASK|FTT_VAR_ID_TYPE_MASK)) )
#define FTT_VAR_ID_CMP_ID_TYPE_TAG( _id1, _id2)   ( (FTT_VAR_ID)_id1 ==
    (FTT_VAR_ID)_id2 )


/* Asynchronous communication channel - Used for Dbs synch */

#define MASTER_BROADCAST_A_CH_ID  1
#define MASTER_BROADCAST_A_CH_SIZE  120
#define MASTER_BROADCAST_A_CH_MTU 1200

typedef enum
{
  FTT_CMD_SET_NODEID=0,
  FTT_CMD_SET_APP_PNP_IDS,
  FTT_CMD_ADD_MSG,
  FTT_CMD_DEL_MSG,
  FTT_CMD_DEL_OLD_TAG,
```

```
  FTT_CMD_DUMMY
}FTT_COMMANDS;

/** \struct MASTER_ASYNCH
\brief Header data carried in the Trigger Message
*/
typedef struct{
  unsigned short COMMAND_id;   ///< Command id
  unsigned short flags;    ///< Sequence number + command hold
} __attribute__ ((packed)) A_MST_CHANNEL_HEADER;

typedef struct{
  A_MST_CHANNEL_HEADER header;
  unsigned short nodeID;
  unsigned char mac[6];
} __attribute__ ((packed)) A_MST_SET_NODEID;

typedef struct{
  A_MST_CHANNEL_HEADER header;
  unsigned char adding_type; //0-change if possible   1-force as new
  unsigned char mesg_type;
  unsigned short mesg_id;
  unsigned char mesg_id_tag;
  unsigned int mesg_size;
  unsigned int mesg_max_size;
  unsigned short mesg_MTU;
  unsigned int mesg_period;
  unsigned char mesg_production_mac[6];
  unsigned char mesg_producerID[40];  //nodeID
  unsigned char mesg_producers_no;
  unsigned char mesg_producers_max_no;
  unsigned char mesg_consumers_no;  //if 0 broadcast
  unsigned char mesg_consumerID[40];
} __attribute__ ((packed)) A_MST_ADD;

typedef struct{
  A_MST_CHANNEL_HEADER header;
  unsigned char mesg_type;
  unsigned short mesg_id;
} __attribute__ ((packed)) A_MST_DEL;

typedef struct{
  A_MST_CHANNEL_HEADER header;
  unsigned char mesg_type;
  unsigned short mesg_id; //It embedds the old tag
} __attribute__ ((packed)) A_MST_DEL_OLD_TAG;

typedef struct{
  A_MST_CHANNEL_HEADER header;
} __attribute__ ((packed)) A_MST_DUMMY;

typedef struct{
  A_MST_CHANNEL_HEADER header;
  unsigned char app_id;
  unsigned char nodeID;
  unsigned char ack; // 0/1
  unsigned short tx_id;
  unsigned short rx_id;
} __attribute__ ((packed)) A_PNP_IDS_NOTIFY;

#endif   /* ----- #ifndef FTT-GLOBAL2_FILE_HEADER_INC  ----- */
```

A.5 Ports

```
/*****************************************************************************
 * Ports.h:
 *****************************************************************************
 * Copyright (C) 2006-2012 the FTT-SE team.
 *
 * Author: Ricardo Marau <marau at fe.up.pt>
 *
 * FTT-SE is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  FTT-SE is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with FTT-SE.  If not, see <http://www.gnu.org/licenses/>.
 *****************************************************************************/

#ifndef _PORTS_H
#define _PORTS_H

/* This must be the first include in the .c files */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#ifdef RTL_PRO
#include <rtl_pthread.h>
#include <rtl_semaphore.h>
#else
#include <pthread.h>
#include <semaphore.h>
#include <sched.h>
#endif

#ifdef RTL_PRO

#else

#define rtl_pthread_t                    pthread_t
#define rtl_pthread_attr_t               pthread_attr_t
#define rtl_pthread_attr_init            pthread_attr_init
#define rtl_pthread_attr_setschedparam   pthread_attr_setschedparam
#define rtl_pthread_create               pthread_create
#define rtl_pthread_cancel               pthread_cancel
#define rtl_pthread_join                 pthread_join

#define rtl_pthread_mutex_t              pthread_mutex_t
#define rtl_pthread_mutex_lock           pthread_mutex_lock
#define rtl_pthread_mutex_unlock         pthread_mutex_unlock
#define rtl_pthread_mutex_init           pthread_mutex_init
#define rtl_pthread_mutex_destroy        pthread_mutex_destroy
#define rtl_pthread_attr_setdetachstate  pthread_attr_setdetachstate

#define rtl_pthread_cond_t               pthread_cond_t
#define rtl_pthread_cond_wait            pthread_cond_wait
#define rtl_pthread_cond_signal          pthread_cond_signal
#define rtl_pthread_cond_broadcast       pthread_cond_broadcast
#define rtl_pthread_cond_init            pthread_cond_init
#define rtl_pthread_cond_destroy         pthread_cond_destroy
```

```
//#define rtl_pthread_mutexattr_t          pthread_mutexattr_t
//#define rtl_pthread_condattr_init        pthread_condattr_init
//#define rtl_pthread_condattr_destroy     pthread_condattr_destroy
//#define rtl_pthread_mutexattr_setpshared pthread_mutexattr_setpshared
//#define RTL_PTHREAD_PROCESS_SHARED       PTHREAD_PROCESS_SHARED


#define rtl_sem_t                       sem_t
#define rtl_sem_wait                    sem_wait
#define rtl_sem_trywait                 sem_trywait
#define rtl_sem_timedwait               sem_timedwait
#define rtl_sem_post                    sem_post
#define rtl_sem_init                    sem_init
#define rtl_sem_destroy                 sem_destroy

#define RTL_TIMER_ADVANCE 0

#define rtl_clock_nanosleep             clock_nanosleep
#define rtl_clock_gettime               clock_gettime

#define RTL_CLOCK_REALTIME              CLOCK_REALTIME
#define rtl_timespec                    timespec
#define rtl_sched_get_priority_max      sched_get_priority_max
#define rtl_sched_get_priority_min      sched_get_priority_min
#define rtl_sched_param                 sched_param
#define rtl_pthread_attr_setstacksize   pthread_attr_setstacksize
#define rtl_pthread_attr_setstackaddr   pthread_attr_setstackaddr
#define RTL_SCHED_FIFO                  SCHED_FIFO

#define rtl_timespec_add          timespec_add
#define rtl_timespec_add_ns             timespec_add_ns
#define rtl_timespec_from_ns            timespec_from_ns
#define rtl_timespec_gt         timespec_gt
#define rtl_timespec_sub          timespec_sub
#define rtl_timespec_sub_ns       timespec_sub_ns
#define rtl_timespec_getns        timespec_getns
#define RTL_TIMER_ABSTIME               TIMER_ABSTIME

#define rtl_shm_open                    shm_open
#define rtl_mmap                        mmap
#define rtl_munmap                      munmap
#define rtl_close                       close
#define rtl_shm_unlink                  shm_unlink

#define RTL_PROT_READ                   PROT_READ
#define RTL_O_CREAT                     O_CREAT
#define RTL_O_RDWR                      O_RDWR
#define RTL_O_RDONLY                    O_RDONLY
#define RTL_O_PHYS                      0
#define RTL_PROT_WRITE                  PROT_WRITE
#define RTL_MAP_SHARED                  MAP_SHARED
#define RTL_MAP_FAILED                  MAP_FAILED

#define rtl_errors            errors
#define rtl_combinations          combinations
#define rtl_next_combination        next_combination
#define rtl_number_combinations     number_combinations
#define rtl_factorial         factorial

#endif

#ifdef RTL_PRO
#define ERROR_MSG(F, S...)     {rtl_printf("\n[%s] %s() " F , __FILE__,
    __FUNCTION__, ## S);}
#define PRINT_MSG(F, S...)     {rtl_printf("\n" F , ## S);}
```

```
#define PRINTF(F, S...)        {rtl_printf( F , ## S);}
#else
#define ERROR_MSG(F, S...)     {fprintf(stderr,"\n[%s] %s() " F , __FILE__,
    __FUNCTION__, ## S);fflush(stdout);}
#define PRINT_MSG(F, S...)     {fprintf(stdout,"\n" F , ## S);fflush(stdout);}
#define PRINTF(F, S...)        {fprintf(stdout, F , ## S);fflush(stdout);}

static inline struct timespec timespec_add(struct timespec a, struct timespec b)
{
#define NSEC_PER_SEC  1000000000L
  struct timespec temp;

  if ((a.tv_nsec+b.tv_nsec) > NSEC_PER_SEC) {
    temp.tv_sec = a.tv_sec + b.tv_sec + 1;
    temp.tv_nsec = a.tv_nsec + b.tv_nsec - NSEC_PER_SEC;
  }else{
    temp.tv_sec = a.tv_sec + b.tv_sec;
    temp.tv_nsec = a.tv_nsec + b.tv_nsec;
  }
  return temp;
}

static inline void timespec_add_ns(struct timespec *a, unsigned long ns)
{
#define NSEC_PER_SEC  1000000000L
  unsigned long long temp;
  temp = ns + a->tv_nsec;
  while(temp >= NSEC_PER_SEC) {
    temp -= NSEC_PER_SEC;
    a->tv_sec++;
  }

  a->tv_nsec = temp;
}

static inline struct timespec timespec_from_ns(unsigned long ns)
{
  struct timespec x={0,0};
  timespec_add_ns( &x, ns);
  return x;
}

static inline unsigned char timespec_gt(struct timespec ts1, struct timespec
    ts2){
  if(ts1.tv_sec  < ts2.tv_sec) return 0;
  if(ts1.tv_sec == ts2.tv_sec) return (ts1.tv_nsec > ts2.tv_nsec);
  if(ts1.tv_sec  > ts2.tv_sec) return 1;
}

static inline struct timespec timespec_sub(struct timespec start, struct
    timespec end)
{
  struct timespec temp;

  if ((end.tv_nsec-start.tv_nsec)<0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
    temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;

  }else{

    temp.tv_sec = end.tv_sec-start.tv_sec;
    temp.tv_nsec = end.tv_nsec-start.tv_nsec;
  }

  return temp;
}
```

```
static inline unsigned long long timespec_getns(struct timespec ts)
{
  return (ts.tv_nsec + 1000000000*(unsigned long long)ts.tv_sec);
}

static inline void timespec_sub_ns(struct timespec *a, unsigned long ns)
{
       while(ns >= NSEC_PER_SEC) {
               ns -= NSEC_PER_SEC;
               a->tv_sec--;
       }

       if(ns > a->tv_nsec){
               a->tv_sec--;
               ns -= a->tv_nsec;
               a->tv_nsec = NSEC_PER_SEC - ns;
       }else{
               a->tv_nsec -= ns;
       }
}

#endif

#ifdef RTL_PRO
//#if __BYTE_ORDER == __BIG_ENDIAN
//cannot use this MACRO. It is not provided by rtl-gcc
inline static unsigned short ntohs (unsigned short nValue){
  return (((nValue>> 8)) | (nValue << 8));
}

inline static unsigned short htons (unsigned short nValue){
  return (((nValue>> 8)) | (nValue << 8));
}

inline static unsigned int ntohi (unsigned int nValue){
  return ((nValue>>24) | ((nValue>>8)&0x0000FF00) | ((nValue<<8)&0x00FF0000) |
       (nValue<<24));
}

inline static unsigned int htoni (unsigned int nValue){
  return ((nValue>>24) | ((nValue>>8)&0x0000FF00) | ((nValue<<8)&0x00FF0000) |
       (nValue<<24));
}
#else
#include <netinet/in.h>
#define ntohi ntohl
#define htoni htonl
#endif

typedef struct{
  rtl_sem_t semaphore;
  unsigned char is_destroyed;
}FTT_SEMAPHORE;

static inline signed char FTT_SEMAPHORE_wait( FTT_SEMAPHORE *sem )
{
  signed char ret;

  ret = rtl_sem_wait(&(sem->semaphore));
  if (ret<0) return ret;
  if (sem->is_destroyed) return -10;

  return 0;
}
```

```
static inline void FTT_SEMAPHORE_post( FTT_SEMAPHORE *sem )
{
  rtl_sem_post(&(sem->semaphore));
}

static inline signed char FTT_SEMAPHORE_init( FTT_SEMAPHORE *sem, int
    init_state )
{
  sem->is_destroyed = 0;
  return rtl_sem_init( &(sem->semaphore), 0, init_state);
}

static inline signed char FTT_SEMAPHORE_destroy( FTT_SEMAPHORE *sem )
{
  sem->is_destroyed = 1;
  rtl_sem_post(&(sem->semaphore)); // wake it
  /* and make sure to clean up the semaphore */
  rtl_sem_destroy(&(sem->semaphore));
  return 0;
}
#ifdef RTL_PRO
/* It seams that the  rtl_sem_trywait doesn't work properly under RTL
 * Maybe because it's being called from within the IRQ
 * If I call it right after the sem_init it works.
 * The workaround checks directly the sem value */
static inline signed char FTT_SEMAPHORE_trywait( FTT_SEMAPHORE *sem )
{
  if (sem->semaphore.value){

    sem->semaphore.value --;
    return 0;

  }else{
    return -1;

  }
}
#else
static inline signed char FTT_SEMAPHORE_trywait( FTT_SEMAPHORE *sem )
{
  if ( rtl_sem_trywait(&(sem->semaphore)) < 0 )
    return -1;
  return 0;
}
#endif
```

```
struct errors {
  int num_combinations;
  int combinations[35][7];
};

static inline int factorial ( int n )
{
  int cumul = 1;
  while ( n > 0 ) {
    cumul *= n;
    n--;
  }
  return cumul;
}

static inline int number_combinations ( int k, int i )
{
  return ( (factorial ( k )) / ( factorial ( i ) * factorial ( k - i ) ) );
}
```

```
static inline unsigned char next_combination( unsigned char
    *present_combination, unsigned char index, unsigned char k, unsigned char
    errors, unsigned char end)
{
  if (!end) {
    if (present_combination[0] != (k - errors)) {
          present_combination[index] = present_combination[index] + 1;
              //Cogemos el siguiente elemento posible
          if (index > 0) {
            if (present_combination[index] > k - (errors - index)) { //Si nos
                hemos pasado de
                end = next_combination(present_combination, index - 1, k,
                    errors, 0);
                present_combination[index] = present_combination[index - 1] + 1;
            } else {
                end = 1;
          }
          } else {
            end = 1;
          }
    } else {
      end = 1;
    }
   }
  return end;
}

static inline void combinations ( int k, int num_errors, struct errors *error )
{
  error->num_combinations = number_combinations ( k , num_errors);
  int i;
  int j;
  int l = 0;
  unsigned char present_combination[num_errors];
  for ( i = 0; i < num_errors; i ++ ){
    present_combination[i] = i;
  }
  if ( num_errors == 0 ){
    for ( i = 0 ; i < k ; i++ ){
      error->combinations[0][i] = 1;
    }
  } else {
    for ( i = 0; i < error->num_combinations; i++ ){
      for ( j = 0; j < k; j++ ){
        if ( present_combination[l] == j) {
          error->combinations[i][j] = 0;
          l ++;
        } else {
          error->combinations[i][j] = 1;
        }
      }
      l = 0;
      next_combination( present_combination, num_errors - 1, k, num_errors, 0);
    }

  }
}
```

```
#endif
```

APPENDIX B

PUBLISHED PAPER ABOUT THE RESULTS OF THE PROJECT

The results of this project permitted the writing of a published short paper, in which the candidate participated as a co-author. The other authors of the paper were David Gessner, Alberto Ballesteros, Manuel Barranco and Julián Proenza. The reference of the paper is:

- Gessner, D., Álvarez, I., Ballesteros, A., Barranco, M., Proenza, J., *Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet.* In Proc. 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), September 16-19, 2014, Barcelona, Spain.

Next we can find the complete paper.

Towards an Experimental Assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet

David Gessner, Inés Álvarez, Alberto Ballesteros, Manuel Barranco, Julián Proenza
DMI, Universitat de les Illes Balears, Spain
{davidges, ines.alvarez.91}@gmail.com {a.ballesteros, manuel.barranco, julian.proenza}@uib.es

***Abstract*—The communication subsystem of distributed embedded systems (DES) that must operate continuously and satisfy unpredictable requirement changes must be reliable and flexible. Recently the Flexible Time-Triggered Replicated Star for Ethernet (FTTRS) has been proposed as a communication subsystem that satisfies these two attributes. It is based on the master/multi-slave Flexible-Time Triggered (FTT) communication paradigm and relies on two custom switches, each with its own embedded FTT master. Both masters are active simultaneously and provide the same service. Specifically, they simultaneously and periodically broadcast so-called trigger messages (TMs) in a redundant manner to make them robust to transient channel faults. One of the functions of these TMs is to divide the communication time into rounds called elementary cycles (ECs). For the correct operation of FTTRS, it is important that all slaves agree when each EC starts and ends. A mechanism to achieve this has been recently proposed. This paper presents a first implementation of this mechanism and a series of experimental tests that constitute a first step towards building a prototype of an FTTRS network.**

I. INTRODUCTION

A distributed embedded system (DES), to operate continuously while satisfying unpredictable requirement changes, must be both highly reliable and flexible. To achieve this it requires a communication channel that satisfies those attributes as well. The goal of the *Flexible Time-Triggered Replicated Star for Ethernet* (FTTRS) [1] is to provide such a channel for a project called *Fault Tolerance for Flexible Time-Triggered Ethernet-based systems* (FT4FTT), which aims to provide high reliability and flexibility to all crucial parts of a DES.

FTTRS is based on a switched Ethernet implementation of the Flexible Time Triggered (FTT) communication paradigm [2], a paradigm that provides master/multi-slave communication in a way that allows the communication to adapt to changing real-time requirements. FTTRS attempts to make such communication highly reliable for switched ethernet by using fault tolerance. Its architecture is shown in Figure 1. The main components are two interconnected custom ethernet switches, each of which embeds an FTT master, and a set of FTT slaves connected to both of them.

The embedded masters broadcast a periodic message called *trigger message* (TM), which divides the communication time into rounds of fixed duration called *elementary cycles* (ECs). Specifically, each EC begins with a *trigger message window* (TM window) in which each one of the two embedded masters broadcasts several redundant TMs to the slaves while no other traffic is exchanged on the network. The number of TMs

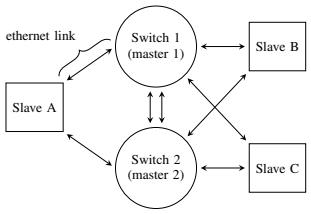


Fig. 1. FTTRS architecture.

broadcast by each master in each EC is given by a parameter k, which is a function of the bit error rate of the channel. Moreover, the broadcasts are synchronized such that when one master transmits its nth TM of a given TM window, the other transmits its nth TM of the same TM window quasi-simultaneously [3]. In other words, the TM transmissions of the two masters occur in lockstep.

For FTTRS to function correctly, the slaves must agree when each EC begins and ends. Since we want FTTRS to be highly reliable, we have recently proposed a mechanism to achieve this even if due to channel faults each slave fails to receive all but one TM per TM window [4]. In this paper we present a first implementation of this mechanism and a series of tests to check that the implementation is correct. Moreover, we provide some first results regarding the viability of achieving a precise EC synchronization in practice with the mechanism.

The remainder of the paper proceeds as follows. Section II summarizes the EC synchronization mechanism used by the slaves. Section III describes our implementation of the EC synchronization mechanism. Section IV describes the tests we performed and the results we obtained. Finally, Section V concludes the paper and points to future work.

II. THE SLAVE EC SYNCHRONIZATION MECHANISM

This section summarizes the slave EC synchronization mechanism that was first presented in a previous paper [4].

As mentioned in the introduction, of the k TM replicas broadcast by each master, the corresponding slave might receive all k replicas or only a subset of them due to transient faults. Regardless of which specific replicas each slave receives on each of its links, the time instants when the slaves consider each EC to start and end must align. This can be achieved by

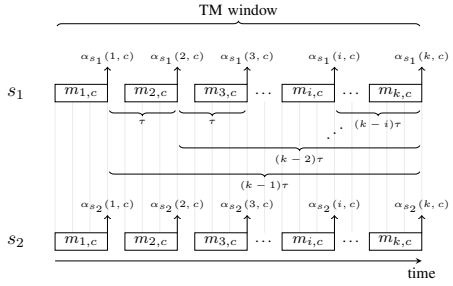


Fig. 2. Alignment of TM arrival times.

the recently proposed EC synchronization mechanism under certain conditions, which we will call *EC synchronization requirements*. These conditions can be summarized as follows: (a) each of the slaves to be synchronized receives at least one TM per TM window; (b) the TMs of a TM window are broadcast such that all slaves that receive the same TM do so at the same time through each of their links; (c) the TMs are broadcast with the same fixed intertransmission time τ; and (d) the amount by which the clocks of the slaves drift apart during one EC is negligible. Under these conditions, the following EC synchronization mechanism can be used for the slaves. (Note that because of condition (b), we do not need to distinguish between the links of each slave.)

Let S_c denote the set of slaves in a given FTTRS network that remain non-faulty at the end of EC c. Moreover, let $m_{i,c}$ denote the TM with sequence number i in EC c, where $i \in \mathbb{N}$ and $1 \leq i \leq k$. Also, let $M_{s,c}$ be the set of TMs that a slave $s \in S_c$ receives during the TM window of an EC c. Furthermore, let $\alpha_s(i,c)$ denote the expected arrival time of $m_{i,c} \in M_{s,c}$ at slave $s \in S_c$. Figure 2 illustrates these expected TM arrival times during the TM window of an EC c for two slaves $s_1, s_2 \in S_c$.

The expected arrival time of $m_{k,c}$ at a slave s is

$$\alpha_s(k,c) = \alpha_s(i,c) + (k-i)\tau, \tag{1}$$

which coincides with the end of the TM window. This is also illustrated in Figure 2 for slaves s_1 and s_2.

If s received at least one TM replica $m_{i,c}$ in c, then (1) can be calculated for all $s \in S_c$ for each EC c. Because of condition (b), in addition we have that

$$\alpha_{s_1}(k,c) = \alpha_{s_2}(k,c), \quad \text{where } s_1, s_2 \in S_c. \tag{2}$$

The EC synchronization mechanism for the slaves can therefore synchronize the end of the TM window of EC c among s_1 and s_2 by using the TM arrival time $\alpha_{s_1}(k,c) = \alpha_{s_2}(k,c)$ as the synchronization event.

III. Implementing the slave EC synchronization

To simplify a first experimental assessment of the EC synchronization mechanism, in our current implementation we abstracted away the presence of two switches and masters. This is a reasonable abstraction for an initial experimental evaluation of the EC synchronization mechanism because according to the FTTRS design the two masters are replica determinate and thus provide identical service from the slaves point of view [3]. The advantage of this abstraction is that it allowed us to test the EC synchronization among slaves without first having to implement the enforcement of master replica determinism, which is required for two masters to transmit their TMs in lockstep. Moreover, it also allowed us to obtain some first results without having to implement how the slaves manage the replication of the TMs caused by having two masters generating sets of k TM replicas.

Regarding the specifics of our implementation, we took as our starting point a software implementation of FTT-SE [5], [6], which is a non-fault-tolerant switched Ethernet implementation of FTT. In this implementation the slaves and masters are implemented as processes to be executed in user space on top of an x86-based computer running a GNU/Linux operating system. In a typical FTT-SE network based on this implementation, several computers running GNU/Linux are interconnected by means of a single commercial off-the-shelf (COTS) Ethernet switch. One of these computers executes the process for the FTT-SE master, and each one of the others executes an FTT-SE slave process.

For our implementation we made several changes to the FTT-SE codebase. First, we modified the code for the master such that it implements the behaviors that are relevant for the FTTRS slave EC synchronization mechanism. Specifically, we made changes for the master to transmit a pre-specified number k of TMs per EC, and to do that with a fixed period τ within each EC. Both k and τ, as well as the desired EC duration, can be passed as arguments to the executable for the master. Moreover, we modified the code for the master to encapsulate k and τ in all TMs, and to add sequence numbers to them. Regarding the EC duration, it was already encapsulated in the TMs in the original FTT-SE codebase.

With respect to the code for the slaves, we modified it to implement the EC synchronization mechanism using a busy wait. When a TM $m_{i,c}$ is received by a slave s at time $\alpha_s(i,c)$, it calculates the time remaining to the end of the current TM window. This time is then added to $\alpha_s(i,c)$ in order to set the absolute time when the TM window in EC c must end. We call this time the *TM window expiration time* of EC c and it coincides with $\alpha_s(k,c)$ of equation (1). If $m_{i,c}$ was the first TM received in c, a thread is created which will, by means of a busy wait, indicate the TM window expiration time. Note that the advantage of using a busy wait is that we avoid sleeping the slave process. This provides better behavior than timers because with timers processes are sent to sleep and must be awoken by the OS, which may cause nondeterministic delays. Finally, as explained in the next section, we added some additional code to the slaves to evaluate the implementation of the mechanism.

IV. Testing the implementation

To test the implementation of the EC synchronization mechanism we used software implemented fault injection (SWIFI) [7]. This means that each slave process executes additional code that forces them to ignore certain TMs. In this way, we can test whether the implementation is robust to TM

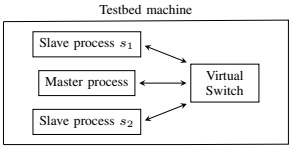


Fig. 3. Virtualized prototype architecture.

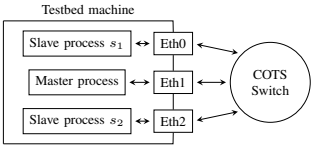


Fig. 4. Physical switch prototype architecture.

losses as foreseen by the design of the EC synchronization mechanism. In particular, the added SWIFI code chooses which TMs must be ignored depending on their sequence number. This is done such that all possible combinations of missing up to $k-1$ TMs on a slave's link are tested, which are all TM loss scenarios under which the designed mechanism can synchronize the slaves. The number of these TM loss combinations is given by

$$\left(\sum_{e=0}^{k-1}\binom{k}{e}\right)^{n}, \tag{3}$$

where k is the number of TMs per EC, e is the number of lost TMs, and n is the number of slaves attached to the network.

In addition to the SWIFI code, we also added instrumentation code to the slaves to timestamp the TM window expiration time of each EC. This allows us to evaluate the precision with which we are able to synchronize the ECs among the slaves.

A. Test setup

To test the slave EC synchronization we used two different setups, but with several commonalities. First, in both setups we used a single master and two slaves. This is the minimum number of slaves and masters required for a first experimental evaluation of the slave EC synchronization. Second, in both setups the process for the master and the two processes for the slaves were executed on the same machine and under the same GNU/Linux OS instance. This made it possible for the slaves to share the same clock, providing a common timebase for their timestamping. Moreover, it simplified initializing the communication between the master and the slaves. Finally, in both setups the master process and slave processes are attached to a single 100 Mbps Ethernet switch. Next, we highlight the differences between the two setups.

1) Virtualized network setup: In this setup the single switch is a virtual one and the processes are attached to it through virtual Ethernet interfaces. Specifically, a *virtual distributed Ethernet switch* is used [8]. This setup allowed us to check the performance of the synchronization without taking into account physical Ethernet interfaces, propagation delay, and switching delay. A diagram of this setup can be seen in Figure 3.

2) Shared machine with physical switch: In this setup the single switch is a COTS Ethernet switch and each process is attached to it by means of a different physical Ethernet interface of the testbed machine. A diagram of this setup is shown in Figure 4.

TABLE I. EXPERIMENT PARAMETERS.

	k	τ (μs)	EC length (μs)	# Test runs
Virtual Switch	4	100	1000	1000
COTS switch	4	100	1000	1000

B. Test parameters and results

To evaluate the results we define the *measured EC offset* between two slaves s_1 and s_2 in cycle c as $|t_{s_1}(c) - t_{s_2}(c)|$, where $t_{s_1}(c)$ and $t_{s_2}(c)$ indicate the recorded timestamp in EC c by slave s_1 and s_2, respectively. In other words, we measure by how much the TM window expiration times of the slaves deviate in each EC. Note that since these times correspond to $\alpha_{s_1}(k,c)$ and $\alpha_{s_2}(k,c)$, respectively, the measured EC offset should be zero according to equality (2) of Section II. Under our real experimental conditions, however, the EC synchronization requirements are not perfectly satisfied and the execution time of the slave processes is not deterministic, e.g., the process might be preempted by the OS. Thus, the measured EC offset is a measure of the precision with which the EC synchronization among two slaves is achieved in practice.

Table I shows the parameters we used in our tests. Specifically, the EC length has been set to 1 ms. This is a suitable value for providing timely communications for typical control applications. Regarding the intertransmission time τ with which the master process sends the TMs to the slaves, it must be greater than the transmission time of a TM, including the Ethernet interframe gap (96 bit times). Since during the TM window only TMs are exchanged on the network, this prevents the TMs from being queued in the switch output ports. This helps ensure that the TMs are not only transmitted with an intertransmission time τ, but also reach the slaves with the necessary interarrival time τ. Note that queuing at a switch output port could prevent this by introducing significant non-deterministic delays. This might occur, for instance, if some TMs occupy a link for a longer or shorter time than the expected TM transmission time due to errors in the link such as dribble bits. In our experiments the TMs do not carry scheduling data [2] and thus are only 14 bytes long, which fits within the 46 bytes of data padding of an Ethernet frame of minimum size (72 bytes). Since we are using 100 Mbps Ethernet, we must therefore set τ to a value greater than $(72 \cdot 8 + 96)/100 = 6.72$ μs. We have chosen a value of 100 μs. We have also chosen to perform 1000 test runs, where each test run injects all possible ways of losing TMs in both slaves with $k = 4$. This yields $(\sum_{e=0}^{4-1}\binom{4}{e})^2 = 225$ fault injections per test run according to equation (3), giving us 225000 sample points. This already provided us with important insights, as described in the next paragraph and Section V, and we had no need for testing different parameter sets with our current

TABLE II. MEASURED EC OFFSET RESULTS.

	# samples	mean (μs)	std. dev. (μs)	max (μs)
Virtual Switch	225000	1.94	0.84	47.62
COTS switch	225000	0.69	1.36	91.37

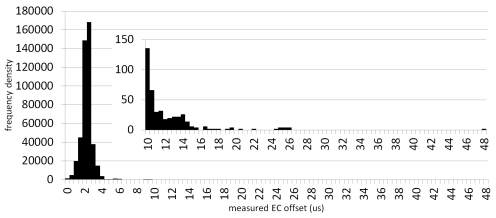


Fig. 5. Histogram of measured EC offset for shared machine with virtual switch. Bin size is 0.5 μs. The superimposed figure is a close-up of the right tail of the histogram.

implementation and experimental setups.

The results are shown in Table II. The mean and standard deviation of the measured EC offset indicate that with both experimental setups the implementation achieves a good EC synchronization, on the order of 0.1–0.2% of the EC length, in most cases. However, the main measure of interest is the maximum measured EC offset. This is so because FTTRS provides real-time communication, where the end of each EC constitutes a hard deadline for the round-based communication to take place. Unfortunately, with the current implementation, and with both experimental setups, we can get values that are significantly larger than the mean: on the order of 5–10% of the EC length. This is also confirmed by the histograms of figures 5 and 6. They both reveal that the distribution of the EC offset has a long tail in the current implementation.

V. CONCLUSIONS AND FUTURE WORK

This paper constitutes a first step towards building a prototype of FTTRS. Specifically, it presents a first implementation of an EC synchronization mechanism for the slaves, which is a key part of FTTRS. The implementation was tested with a single master and two slaves, all executing as processes on the same machine under the same GNU/Linux OS instance. This significantly simplified a first experimental verification of the implementation. The experiments consisted in having the master process transmit its TMs to the slave processes through both a virtual switch and a physical COTS switch. The results of the tests are promising: they show that the implementation achieves a good EC synchronization most of the time with the experimental setups we used. Nevertheless, they also highlighted that occasionally a large EC offset between the slaves can be observed. This, as was to be expected, is due to the slaves being executed as user processes on top of a non-real-time OS. We inherited this from the FTT-SE codebase. If large EC lengths are used, then a maximum EC offset on the order of 50-100 μs, as we have measured, are acceptable. However, with FT4FTT we are also targeting control applications with high sampling rates, where the measured values can be problematic. In this paper we therefore confirmed that the FTT-SE codebase must be further adapted to our needs.

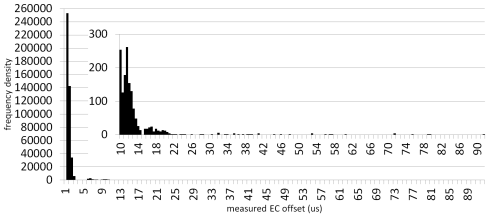


Fig. 6. Histogram of measured EC offset for shared machine with COTS switch. Bin size is 0.5 μs. The superimposed figure is a close-up of the right tail of the histogram.

The next step involves updating the implementation to take advantage of a Linux kernel that offers real-time features. We are considering Xenomai [9] for this. Moreover, we also plan to modify the experimental setup by moving the slave processes and the master process to different machines and measuring the EC offset in this new distributed setup. This will give us a better view of the precision with which ECs can be synchronized among slaves in an actual FTTRS implementation.

Further future work includes implementing the mechanisms to achieve replica determinism for two FTTRS masters and to then test the implementation of the EC synchronization mechanism with two such replica determinate masters.

ACKNOWLEDGEMENTS

This work was supported by project DPI2011-22992 and grant BES-2012-052040 (*Spanish Ministerio de economía y competividad*), and by FEDER funding.

REFERENCES

[1] D. Gessner, J. Proenza, M. Barranco, and L. Almeida, "Towards a flexible time-triggered replicated star for Ethernet," in *Proc. 18th IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, Cagliari, Italy, Sep. 2013.

[2] P. Pedreiras and L. Almeida, "The Flexible Time-Triggered (FTT) paradigm: an approach to QoS management in distributed real-time systems," in *Proc. Int. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2001.

[3] D. Gessner, J. Proenza, and M. Barranco, "A Proposal for Master Replica Control in the Flexible Time-Triggered Replicated Star for Ethernet," in *Proc. 10th IEEE Int. Workshop on Factory Communication Systems (WFCS)*, Toulouse, France, May 2014.

[4] ——, "A Proposal for Managing the Redundancy Provided by the Flexible Time-Triggered Replicated Star for Ethernet," in *Proc. 10th IEEE Int. Workshop on Factory Communication Systems (WFCS)*, Toulouse, France, May 2014.

[5] R. Marau, L. Almeida, and P. Pedreiras, "Enhancing real-time communication over COTS Ethernet switches," in *Proc. 6th IEEE Int. Workshop on Factory Communication Systems (WFCS)*. Torino, Italy: IEEE, 2006, pp. 295–302.

[6] (2014, May) FTT-SE v2.6.2 source code. [Online]. Available: http://paginas.fe.up.pt/~ftt/repository/ftt-se.2.6.2.tar.bz2

[7] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[8] R. Davoli, "Vde: Virtual distributed ethernet," in *Proc. 1st IEEE Int. Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities.* IEEE, 2005, pp. 213–220.

[9] P. Gerum, "Xenomai—Implementing a RTOS emulation framework on GNU/Linux," *White Paper, Xenomai*, 2004. [Online]. Available: http://www.xenomai.org/documentation/xenomai-head/pdf/xenomai.pdf

BIBLIOGRAPHY

[1] H. Kopetz, *Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997. 1.1

[2] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers, April 2009. 1.1

[3] B. Bouyssounouse and J. E. Sifakis, "Embedded Systems Design. The ARTIST Roadmap for Research and Development," *Lecture Notes in Computer Science*, vol. 3436, 2005. 1.1

[4] L. Almeida, P. Pedreiras, and J. A. G. Fonseca, "The FTT-CAN protocol: why and how," *Industrial Electronics, IEEE Transactions*, vol. 49, no. 6, pp. 1189 – 1201, december 2002. 1.1, 2.1, 2.1

[5] FlexRay™, "FlexRay Communications System - Protocol Specification, Version 2.1," 2005. 1.1

[6] P. Pedreiras, P. Gai, L. Almeida, and G. C. Buttazzo, *FTT-Ethernet: a flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems. IEEE Transactions on Industrial Informatics*, 2005, 1, 162-172. 1.1

[7] R. Marau, L. Almeida, and P. Pedreiras, "Enhancing real-time communication over COTS Ethernet switches," in *Proc. 6th IEEE Int. Workshop on Factory Communication Systems (WFCS)*. Torino, Italy: IEEE, 2006, pp. 295–302. 1.1, 1.3, 3, 8.1, A

[8] (2014) Fault Tolerance for Flexible Time-Triggered project page. [Online]. Available: http://srv.uib.es/project/16 1.1, 8

[9] D. Gessner, J. Proenza, M. Barranco, and L. Almeida, "Towards a flexible time-triggered replicated star for Ethernet," in *Proc. 18th IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, Cagliari, Italy, Sep. 2013. 1.1, 2.3, 5.1.2, 8.2

[10] D. Gessner, J. Proenza, and M. Barranco, "A Proposal for Managing the Redundancy Provided by the Flexible Time-Triggered Replicated Star for Ethernet," in *Proc. 10th IEEE Int. Workshop on Factory Communication Systems (WFCS)*, Toulouse, France, May 2014. 1.1, 1.3, 8.1

[11] (2014, May) FTT-SE v2.6.2 source code. [Online]. Available: http://paginas.fe.up.pt/~ftt/repository/ftt-se.2.6.2.tar.bz2 1.3, 3, 8.1, A

[12] P. Pedreiras, P. Gai, L. Almeida, and G. C. Buttazzo, "FTT-Ethernet: a flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 3, pp. 162–172, 2005. 2.1, 2.2

[13] R. Marau, "Real-time communications over switched Ethernet supporting dynamic QoS management," Ph.D. dissertation, 2009. [Online]. Available: http://ria.ua.pt/handle/10773/2224 2.1

[14] R. Santos, "Enhanced Ethernet Switching Technology for Adaptive Hard Real-Time Applications," Ph.D. dissertation, Universidade Aveiro, 2011. 2.1

[15] R. Davoli, "Vde: Virtual distributed ethernet," in *Proc. 1st IEEE Int. Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities*. IEEE, 2005, pp. 213–220. 5.1.2

[16] W. Stallings, *Data and Computer Communications*. Prentice Hall, 2011. (document), 5.2

[17] A. Ballesteros, J. Proenza, D. Gessner, G. Rodriguez-Navas, and T. Sauter, "Achieving Elementary Cycle Synchronization between Masters in the Flexible Time-Triggered Replicated Star for Ethernet," in *Proc. 19th IEEE 19th IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, Barcelona, Spain, Sep. 2014. 6.1.1, 8.2

[18] D. Gessner, I. Álvarez, A. Ballesteros, M. Barranco, and J. Proenza, "Towards an experimental assessment of the Slave Elementary Cycle Synchronization in the Flexible Time-Triggered Replicated Star for Ethernet," in *Proc. 19th IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, Barcelona, Spain, Sep. 2014. 8.3