Universidade de Aveiro
2010
Departamento de Electrónica Telecomunicações e Informática

Filipe Miguel Santos Teixeira

Sistema de processamento paralelo aplicado ao segmento automovel

Parallel processing system applied to the automotive segment

Universidade de Aveiro
2010
Departamento de Electrónica Telecomunicações e Informática

Filipe Miguel Santos Teixeira

Sistema de processamento paralelo aplicado ao segmento automóvel

Parallel processing system applied to the automotive segment

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações (M.I.E.E.T.), realizada sob a orientação científica do Prof. Doutor Manuel Bernardo Salvador Cunha e do Prof. Doutor José Luís Costa Pinto Azevedo, Professores Auxiliares do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho à minha esposa e filho pelo incansável apoio. Ao Pedro Kulzer que me possibilitou fazer parte deste projecto. Ao Professor Bernardo Cunha pelos aconselhamentos técnicos. E um agradecimento especial ao resto da equipa ECU2010, Paulo, Nelson e Rui pelo óptimo espírito de equipa e por terem feito do projecto ECU2010 o meu melhor momento profissional.

O júri

Presidente	Prof. Dr. Tomas António Mendes e Silva Professor do Departamento do DETI
Arguente	Prof. Doutor Sérgio Adriano Fernandes Lopes Professor Auxiliar no Departamento de Electrónica Industrial da Universidade do Minho
Orientador	Prof. Doutor Manuel Bernardo Salvador Cunha Professor Auxiliar do Departamento do DETI
Co-orientador	Prof. Doutor José Luís Costa Pinto de Azevedo Professor Auxiliar do Departamento do DETI

Palavras-chave

Sistemas Embebidos, Processamento Paralelo, Sistemas Integrados, Comunicações, FPGA, Xilinx, Spartan 3E

Resumo

Os actuais sistemas de controlo das funções de motronica

O trabalho apresentado nesta dissertação está integrado no projecto ECU2010, que tem por objectivo o desenvolvimento de um sistema de processamento paralelo escalavel e do software de apoio á sua utilização, para ser aplicado ao controlo de um motor de combustão para o desporto automóvel.

Neste trabalho será desenvolvido a estrutura de hardware paralelo e os sistemas de comunicação entre os nós de processamento paralelo. Tambem faz parte deste trabalho o desenvolvimento de APIs que seram utilizadas pelo software de desenvolvimento para distribuição das funções pelos módulos, bem como implementação de funcionalidades de teste do sistema.

Keywords

Embedded systems, Integrated systems, Parallel Processing, communications, FPGA, Xilinx, Spartan 3E

Abstract

The work presented in this thesis was part of the project ECU2010, the aim of the project was to develop a scalable Parallel Processing System to control a combustion engine to use in Motorsport. It was also an aim of this project to develop a Software Development Environment that enables the user to implement and deploy the motronic functions on the parallel system and debug them on the hardware.

My work focused on two parts, the first part was the idealization of the Parallel Processing System structure and implementation of some of the communication blocks. The second part was implementation an API for the Graphical Interface that would allow an easy handling of the hardware system. My work also focused on the motronic functions distribution and data routing between the modules that make the parallel system.

Table of contents

Chapter 1 Introduction 3
1.1 Summary 3
1.2 Background 3
1.3 Motivation 3
1.4 Goals of this Thesis 4
1.5 Outline of this Thesis 4
Chapter 2 Analysis of possible solutions 5
2.1 Summary 5
2.2 Current Solutions 5
2.2.1 RTI & RapidPro Series 5
2.2.2 ETAS 5
2.2.3 FastPRO 6
2.3 ECU2010 Solution 7
Chapter 3 Design and Implementation 9
3.1 Summary 9
3.2 Development Tools 9
3.3 The Module 10
3.3.1 Configuration Manager 11
3.3.2 Intermodule Communication Manager 13
3.3.3 Gateway Intelligent Memory (GIMy) 15
3.3.4 Log Manager 18
3.3.5 Peripheral Manager 19
3.4 Support Software and API for the graphical interface 23
3.4.1 Allocation of the Functions in the modules 24
3.4.2 Distribution of the Peripherals 25
3.4.3 Programming the ECU and the Peripheral 25
3.4.4 Topology finder 26
3.4.5 Serial communication detector 26
3.4.6 Node Generator 28
3.4.7 Live debug 29
3.4.8 Project lists 30
3.4.9 Display tab 34
Chapter 4 Results 37
4.1 Summary 37
4.2 Module implementation 37
4.3 Running Prototype 38
Chapter 5 Conclusions 41
5.1 Summary 41
5.2 Improvements and Future work 41

Table of figures

Figure 1 - Example of dSpace hardware platform 5
Figure 2 - ETAS software development and hardware targeting overview 6
Figure 3 - "FastPRO" software and hardware combination 6
Figure 4 - Module Ring Topology 7
Figure 5 - Digilent development kit 10
Figure 6 - Module internal architecture 11
Figure 7 - Configuration Manager Architecture 12
Figure 8 - Inter Module Communication Manager 14
Figure 9 - Intermodule Communication Signals 14
Figure 10 - Serial Communication routing table entry 15
Figure 11 - GIMy Interface 16
Figure 12 - Write Operation 17
Figure 13 - Read Operation 17
Figure 14 - Log Table entry format 18
Figure 15 - Log Manager 19
Figure 16 - Peripheral Manager 20
Figure 17 – Gimy2Per Table 21
Figure 18 - Per2GimyTable 22
Figure 19 - Topology finder 26
Figure 20 - Node Generator in function view 28
Figure 21 - Nodes tab overview 31
Figure 22 - Node tab context menu 31
Figure 23 - Module Tab 32
Figure 24 - Functions List tab 32
Figure 25 - Instructions List 33
Figure 26 - Peripheral Tab 34
Figure 27 - Display Tab 35
Figure 28 - Module FPGA Utilization Summary 37
Figure 29 - Mono-cylinder engine testing platform 38
Figure 30 - ECU from different views (3 modules) 39
Figure 31 - Peripherals (Sensors and actuators) 39

Chapter 1
Introduction

1.1 Summary

In this chapter the thesis background is presented, followed by motivation and an overview of the objectives. It's completed by a brief description of the thesis organization.

1.2 Background

The automotive industry is very competitive, even more when it comes to Motorsport, so there are constant investigations in new ways to improve all the components take make up a complete automotive system; one of these components is the Electronic Control Unit (ECU). The ECU is responsible for the execution of the control algorithms of the vehicle, these algorithms or functionalities can be to improve safety, power, comfort, fuel consumption, etc… The execution of such algorithms requires processing power and storage memory. With even more functions inside the ECU, the more processing and storage space is required. Another aspect is that the algorithms normally have inputs and outputs, this is, they might use values from sensors and send orders to actuators.

1.3 Motivation

The automotive industry, especially in Motorsports, trys to develop new functionalities that can put them one step in front of the competition. This requires a system easy to program and debug, but also capable of accommodate the new functionalities when they are developed, this could means an increase of processing power, storage space and interface to sensors and actuators. In current ECUs this can mean that a over dimensioned hardware platform might be used to accommodate new developments, and when that platform is not capable of handling the new functionalities a new hardware platform must be developed. This can cause delays in development, but also increase the cost of the system development.
With this in mind, a parallel processing system capable of escalating depending on the functionalities implemented, that could provide an easy to use programming and debugging interface, could be the solution. This is the motivation for the work presented in this thesis.

1.4 Goals of this Thesis

This thesis demonstrates the work developed as part of the project ECU2010. The ECU2010 was a joint project between Bosch Motorsport, the University of Aveiro and KulzerTEC. Its aims was to develop a new approach in the architecture of a system capable of controlling an internal combustion engine (commonly known as ECU) and also develop the necessary PC software to allow an easy development and deployment of the *motronic* functions and algorithms needed to control the engine. The project objective apart from trying to be deployed in a real racing car in the year 2010, was to make a proof-of-concept.

This thesis will present the work developed in the field of the parallel processing system, and the software APIs that distribute and program the motronic function in the parallel modules.

1.5 Outline of this Thesis

This section shortly summarizes the content of each of the remaining parts of this thesis.

Chapter 2 describes several solutions to the problem and presents the chosen solution.
Chapter 3 describes the implementation of the chosen solution.
Chapter 4 presents the obtained results.
Chapter 5 concludes the whole report presenting the conclusions obtained and presenting some future work.

Chapter 2
Analysis of possible solutions

2.1 Summary

As mentioned before, the automotive industry is in constant development and improvement, which requires development systems that allow the developer to implement and test the control algorithms in the smallest amount of time possible. Currently there are several solutions for this issue, in this chapter some of current available solutions are presented, and in the end the ECU2010 concept is presented.

2.2 Current Solutions

There are several solutions available in the market that allow a fast prototype and implementation of control systems for the automotive industry.

2.2.1 RTI & RapidPro Series

dSpace [*1*] is a company that produces hardware (Figure 1) to integrate in the Matlab Simulink graphical programming environment. It offers a complete HIL (Hardware-In-The-Loop) and vehicle simulation software based on Matlab. Its hardware solution is mostly based in FPGAs, but it also has integration of standard processors. It provides also "RapidPro" extension modules, which are extension modules that can be connected via a proprietary optical-fiber links, allowing to create highly interconnected multi-core systems.

Figure 1 - Example of dSpace hardware platform

2.2.2 ETAS

The ASCET system is provided by ETAS [*2*], it is a graphical programming software that integrates in the hardware platforms also provided by ETAS.

This solution of software and hardware components permits simulation as well as rapid-prototyping at the graphical level.
The hardware apart from having standard microcontrollers, is has also a FPGA to accelerate the digital processing of programs. “Rapid-Prototyping” means that most changes are made in the graphical interface and compiled/downloaded to the hardware target in a matter of minutes.

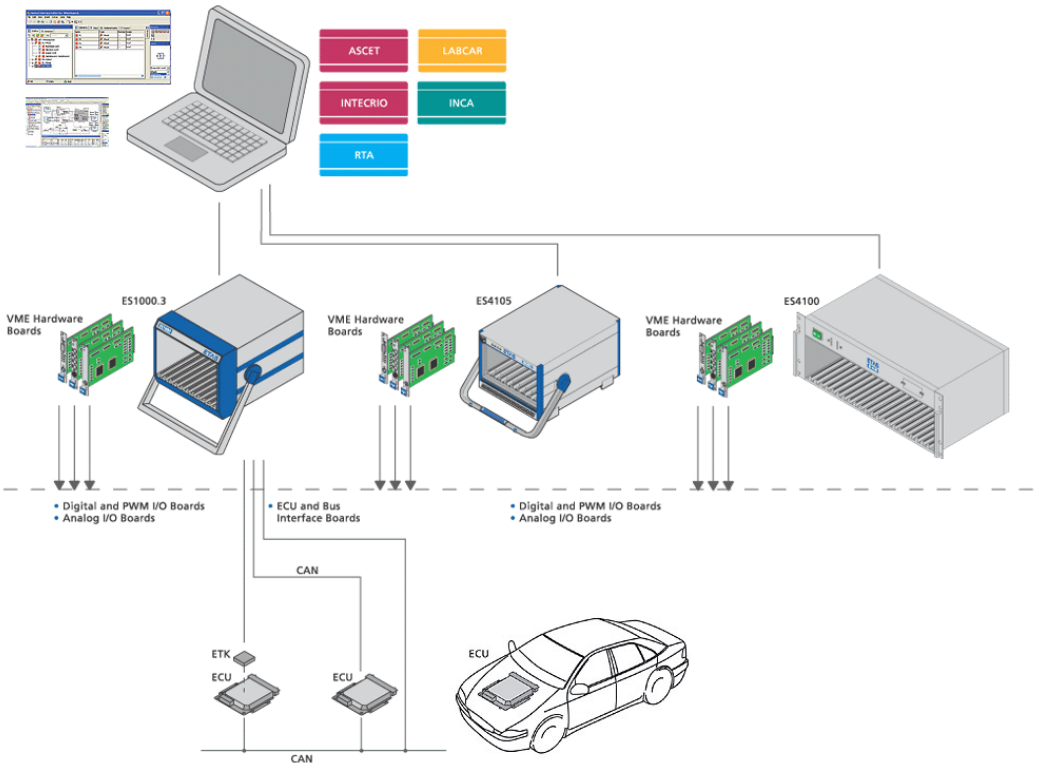

Figure 2 - ETAS software development and hardware targeting overview

2.2.3 FastPRO

FastPRO [3] is made by Magneti-Marelli, it is a hardware and software target combination that allows similar functionalities as Matlab Simulink combinations. It is also base in auto-code generators which translate the graphical programs into hardware-understandable code.
This solution was used for years in the Toyota Formula 1 racing team.

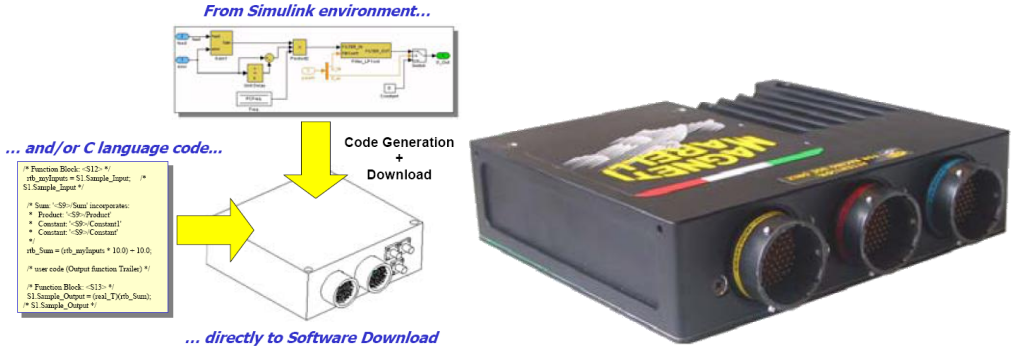

Figure 3 - "FastPRO" software and hardware combination

2.3 ECU2010 Solution

The ECU2010 project should provide a solution to requisites of the development companies and also racing teams.
Some of those requisites are:

- Fast development and hardware programming time
- Scalable processing power and storage space without the need to change completely the hardware platform.
- Easy integration of sensors and actuators.
- Measurement and analysis tools of data logs.

To accomplish these requisites a solution based in parallel processing was developed. The architecture proposed by the ECU2010 to accomplish these requirements is based on a parallel processing system composed of one or more equal modules connected by a serial link, in a ring topology. Also each one of the module has a peripheral bus, where sensors and actuators can be connected. The modules are independent of each other, each one having its processor, storage memory and log capability. The serial link between the modules is used to transfer variables needed across multiple modules. With this architecture if more processing power, storage space or peripherals are needed in the system, we would only have to insert one or more modules in the serial ring, Figure 4.
It was also part of the ECU2010 project the development of an Integrated Development and Management System (IDMS), which is a single application to allow the development, deployment and debug of the functions in the hardware platform. Throughout the complete cycle of the development of an ECU software, there would only be the need for this tool, from the laboratory to the racing track.

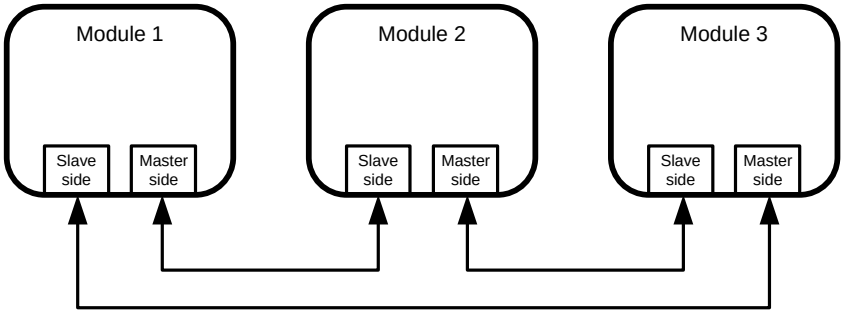

Figure 4 - Module Ring Topology

Chapter 3

Design and Implementation

3.1 Summary

In this chapter, the internal module architecture and the parallel processing allocation will be explained. The blocks that are used to build the module are explained in detail, and the way they communicate. Also the work done in C# to automatically program the modules and perform the function allocation is explained. Finally, some brief explanations on C# controls made to integrate in the IDMS are shown.

3.2 Development Tools

After some research and initial attempts in the implementation of the module using microcontrollers, there was the need to increase the communications and processing speed in each module. With this in mind, the choice for configurable hardware chips or Field-Programmable Gate-Arrays (FPGAs) was taken after some research.
So there was the need to choose a FPGA manufacturer, after some research the XILINX FPGA chips was decided, in detriment of other manufacturers such as ACTEL, ALTERA and LATTICE. XILINX presented the best hardware/software package and is currently regarded as the main FPGA supplier in the world with over 50% market share.
The XILINX FPGA used was the XC3S1600E [*3*] with a total of 1.6 mega-gate. The company DIGILENT provides a development kit, shown in Figure 5 that comes with external RAM, FLASH, connectors, power-supply, optional debugging display, etc.
To develop the IDMS, the .NET platform and the C# development language was used. This decision came from the easy to use development environment and the possible easy transition of the IDMS to PDAs or handheld devices.

To summarize, the tools used to develop this thesis where:

- Spartan-3E FPGA [*8*] development kits [*4*]
- ISE Design Suite to program and configure the FPGAs [*5*]
- ChipScope Pro 12.1 to debug the FPGAS [*6*]
- Microsoft Visual Studio 2005 [*9*] to develop the *"IDMS"* for Windows

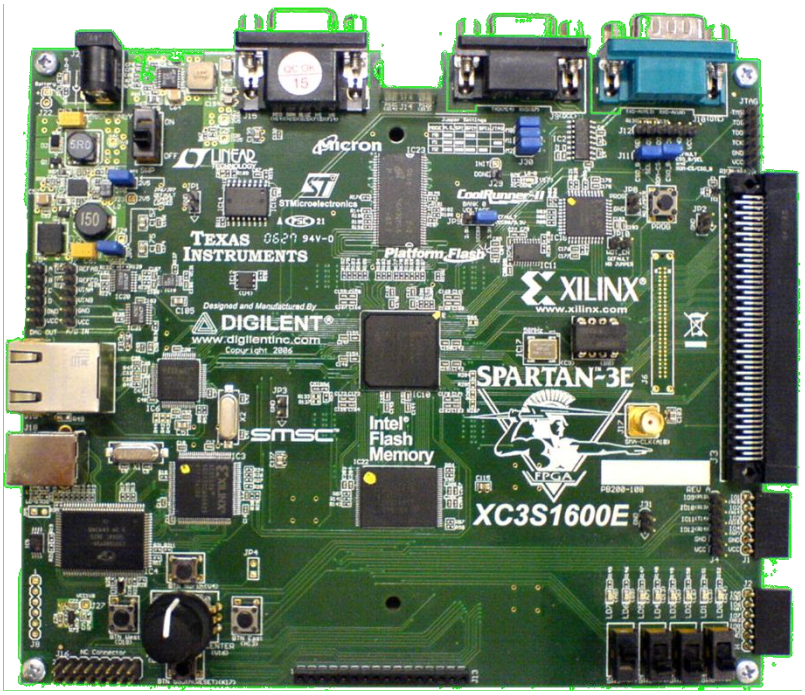

Figure 5 - Digilent development kit

3.3 The Module

As explained before, in this new concept, the ECU is modular, this means that the ECU is made up of individual and equal modules that communicate between each other. The peripherals have the same internal architecture as the ECU module.

Each module must be able to act as a standalone component, this is, a single module ECU must be able to provide the same basic functionality, as an ECU which integrates several modules. The only difference will be in the increase of available resources. The basic functionalities a module must provide are:

- Straightforward communication with USB, TCP/IP, wireless communication or other type of communication platform.
- Processing power, with live prototype capability.
- Peripheral Communication.
- Communication between modules
- Internal and external logging of variables

With this approach, by connecting several modules in the same ECU, there is an increase in communication bandwidth, an increase in processing power by executing functions in parallel, number of peripheral connections available and logging capability.

The internal architecture of the module is shown in Figure 6. The internal architecture is divided in blocks, which are associated with a specific functionality.
Each module executes the *Motronic* functions stored in its local flash memory as fast-as-possible, and uses the Gateway Intelligent Memory (GIMy) to store the variables that are used. The Inter-Module Communication Manager exchanges variables between modules, and routes the variables that are needed in other modules, only the variables that are needed in other modules are transmitted. Then there is the Peripheral Communication that exchanges values with the peripherals connected in the peripheral bus of the module. This communication is also made as fast-as-possible, and all the transmissions are initiated by the module. The Module sends write requests to the peripherals, sending the address and the value to write in the GIMy memory of the peripherals, and sends read request by sending a message with the read command and the address to read, the peripheral then responds with the request.

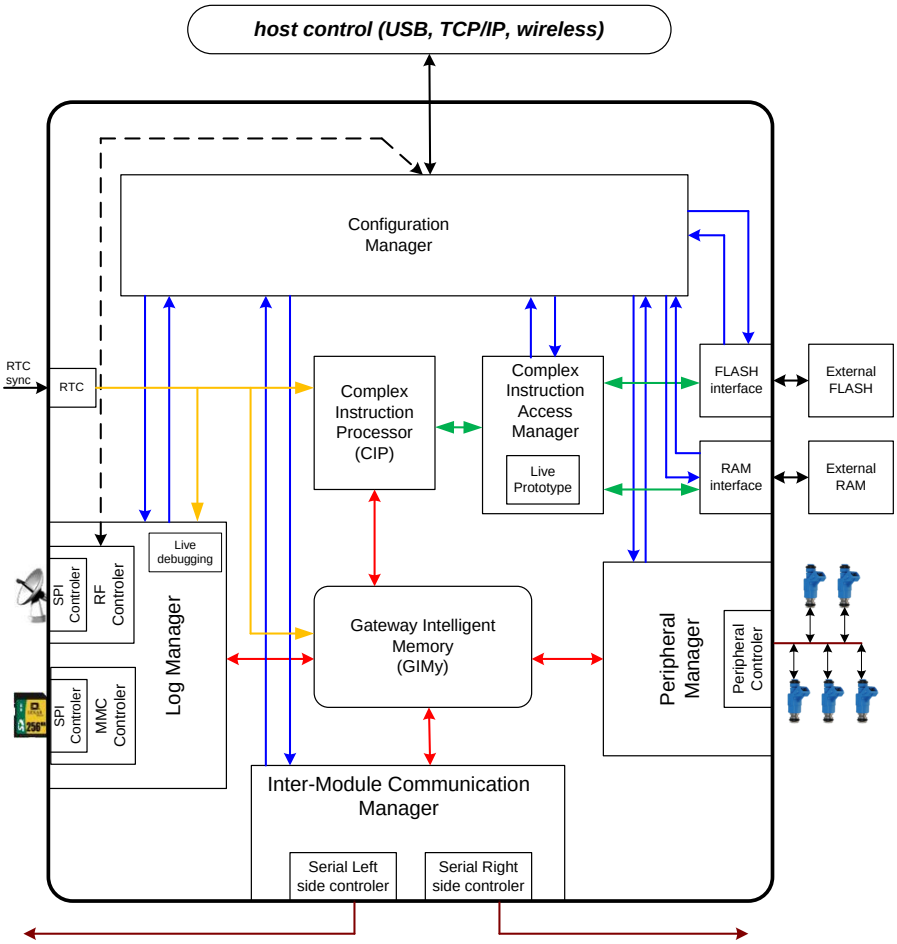

Figure 6 - Module internal architecture

3.3.1 Configuration Manager

The Configuration Manager is responsible by the routing of the messages inside the module. As shown in Figure 7, the Configuration Manager allows communication between the different blocks and the communication protocol used to communicate with

the outside world. In the current implementation the modules use USB to communicate with a computer running the IDMS. There is also an incoming connection from the Flash Interface that allows the loading of the initial configuration of each block after a module reset. This configuration method will be explained in more detail afterwards. As said before the current implementation uses USB, but other protocols can be used if an adapter to the FIFO interface of the configuration manager is developed.

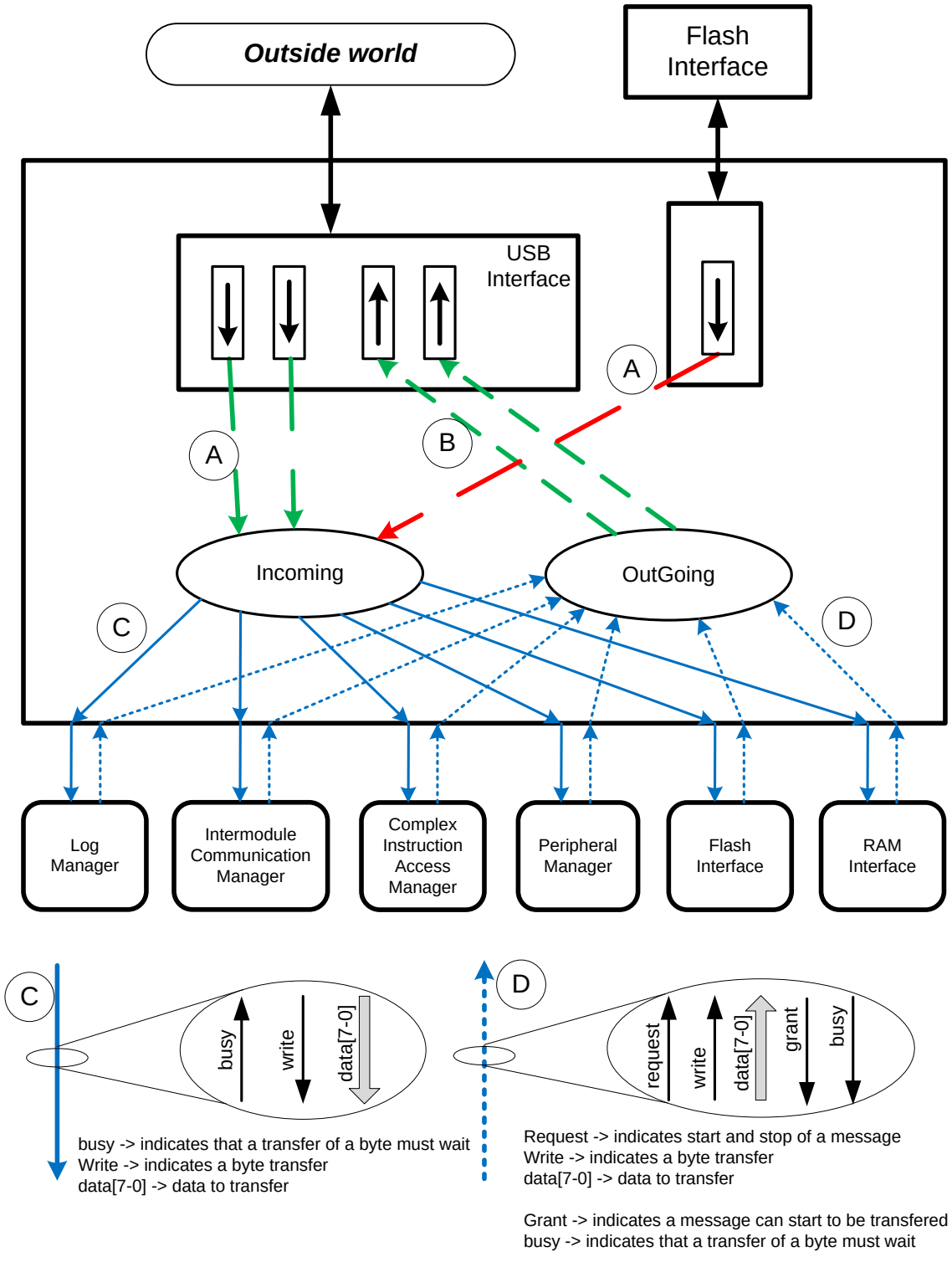

Figure 7 - Configuration Manager Architecture

3.3.2 Intermodule Communication Manager

In every parallel computation system there is the need to exchange data between the different computational blocks, in the implemented system there is the need to exchange values between modules that use the same variable/node. The Intermodule Communication Manager implements the communication between modules. This communication channel is used to exchange the values of the nodes/variables that are used in different modules. To minimize the number of connections from each module to the motherboard, the communication between modules was implemented using a serial communication protocol. The protocol is similar to Serial Peripheral Interface (SPI), but extra signals were added to allow frame synchronization of the communication between the modules. Each module has a master and a slave channel. This allows the implementation of a ring topology, like the one shown in Figure 4; with this topology we can increase the number of modules in a straightforward way.

In Figure 10 the structure of the Inter-Module Communication Manager is presented.
First a brief explanation of the components inside the block:

- *Inter-Module Configuration*: Handles the communication with the Manager Communication. This component allows for the configuration of the routing tables
- *GIMy Access:* Allows the access to the GIMy by the 4 state machines responsible for the serial communications, this is done in a round-robin scheme.
- *Slave TX*: This is a state machine that process the entry's stored in the *slave routing table*. It builds up the data frame that is going to be sent, to the module connected to the slave interface. The frame contains the data to store and the address of the GIMy where to store the data.
- *Slave RX*: This state machine decodes the frames received from the master connected to the slave interface. If the frame is received correctly, then the state machine extracts the address and the BCDFP (Binary Coded Decimal Floating Point) value to write to GIMy and request a write operation.
- *Master TX*: This state machine processes the entries stored in the module *master routing table*. It builds up the data frame that is going to be sent via the module master. The data frame will be received and processed by the *Slave RX* state machine. The frame contains the data to store and the address of the GIMy where to store the data.
- *Master RX*: This state machine decodes the frames received from the slave connected to the master interface. If the frame is received correctly, the state machine extracts the address and the BCDP value to write to GIMy and requests a write operation.

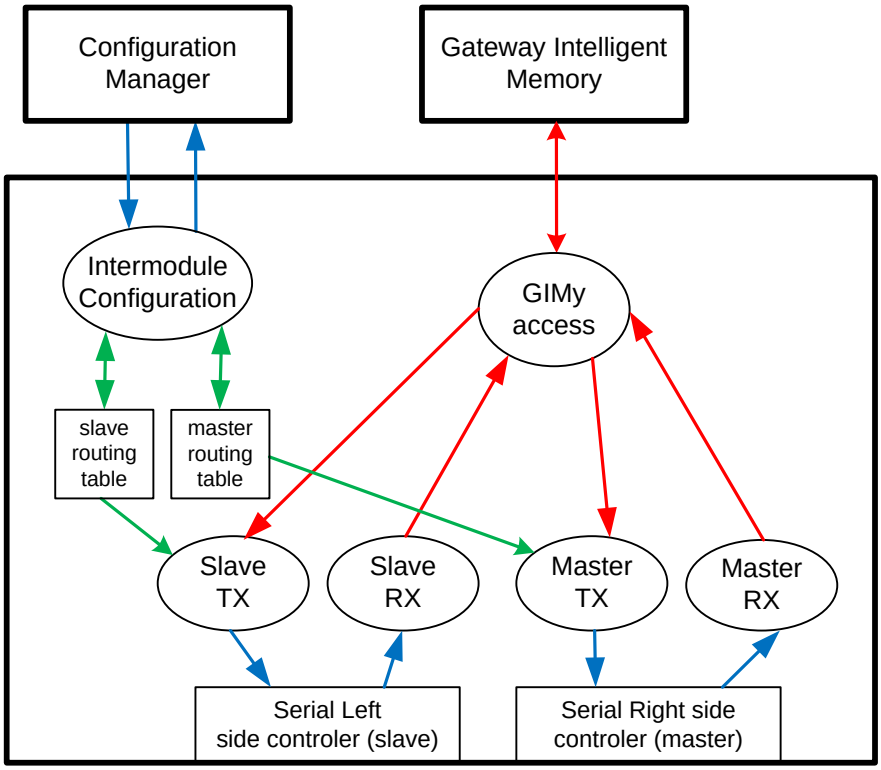

Figure 8 - Inter Module Communication Manager

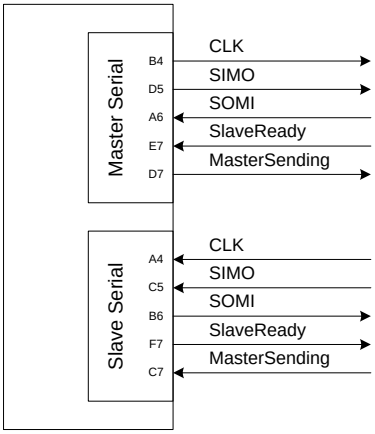

Figure 9 - Intermodule Communication Signals

In Figure 9 the signals used to establish communication between Modules are shown. There are two buses, the Master Serial and the Slave Serial buses.
The Master Serial has the following signals:

- CLK , clock signal generated by the master
- SIMO, data signal driven by the master to send data to the slave
- SOMI, data signal driven by the slave to send data to the master
- Slave Ready, signal that informs the master that the slave is ready to receive data.
- Master Sending, signals driven by the master to inform the slave that the master is sending a new packet

The serial routing table is calculated at the system implementation time, and it is stored in the flash memory and loaded every time a reset occurs. The TX state machines start from address zero of the table and continue until the field *used* is zero. In Figure 10 we show the fields of an entry in the serial routing table. There are three fields:

- *used*: this field indicates if this entry is in use. If the value is one the other fields are taken into account. If the value is zero, then the current values of the fields are taken into account and we start again from address zero of the routing table.
- *Address in the current Module*: This field stores the value of the memory address where the value to send is stored.
- *Address in the next module*: This field contains the memory address where the transmitted value is going to be stored in the next module. This address is sent in the message to the next module and is used by it to store the value of the node; this avoids the need of decoding the address in the receivers end.

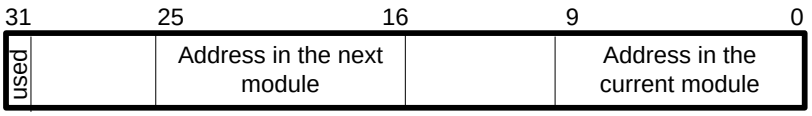

Figure 10 - Serial Communication routing table entry

3.3.3 Gateway Intelligent Memory (GIMy)

The Gateway Intelligent Memory (GIMy) is the internal RAM memory of the module. It stores all the nodes that are processed or pass through the Module via the Intermodule Communications. GIMy has 4 access ports that can be used as read or write ports. It supports multiple reads in the same cycle, but only one write per cycle. If two ports or more ports try to write in the same cycle, there is a handshaking scheme that enables the correct operation. Figure 11 shows the interface of the GIMy block.

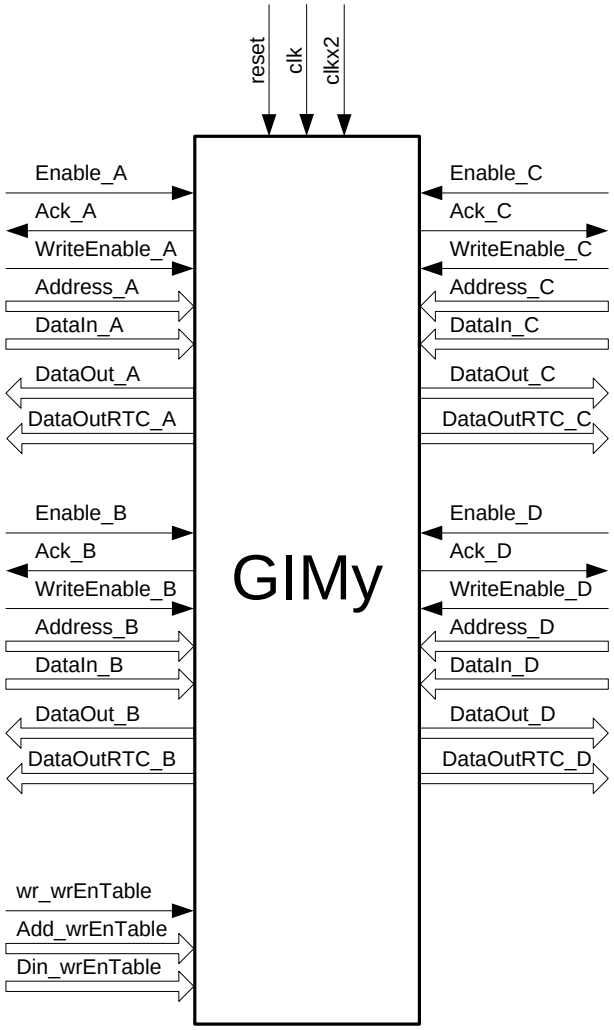

Figure 11 - GIMy Interface

Beside the usual memory signals: write, address, data in and data out, each port has two additional signals, *Enable* and *Ack* signals, these signals are used to perform a handshaking mechanism on the operation between the GIMy and the process that is requesting the access. The *Enable* signal is asserted by the process that is requesting the access to the memory. At the same time that the process asserts the *Enable* signal it must also drive the signals *write*, *address* and *data in* to perform the required operation. D*ata in* is only necessary when the requested operation is a write operation. The signal *Ack* is asserted by GIMy, and signals the process that the clock cycle he requested was granted.

To reset the *Ack* signal, the accessing process must reset the signal *Enable* for at least one clock cycle.

All four access ports can have their *Enable* line asserted simultaneously; it is GIMy responsibility to set the corresponding *Ack* flags to signal the access granted to the processes.

A write operation is shown in Figure 12. The processes accessing GIMy must drive the signals in the rising edge of the CLK signal.

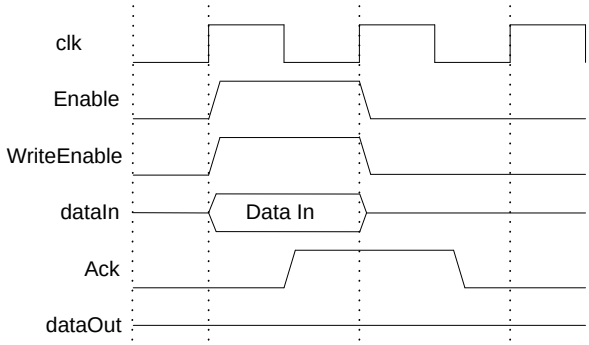

Figure 12 - Write Operation

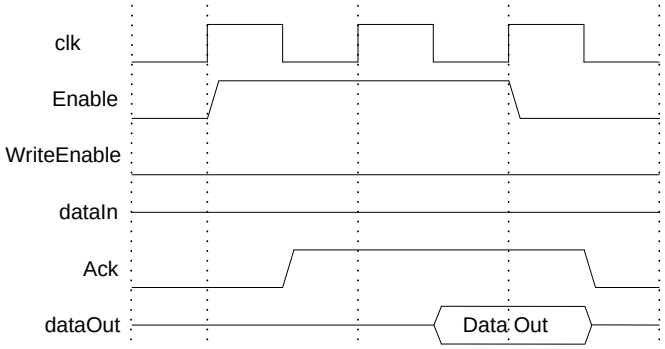

Figure 13 - Read Operation

The GIMy does not only store the value of the node, it also stores additional information for each node. In the current implementation, associated with each memory address a timestamp is also stored. The timestamp is stored each time a write operation is performed.

Each address also has a write enable flag, this flag is only used to enable the writing to each address by two of the four access ports. This way we have two ports that have full access to all memory address and two ports that we can control the write permissions. In the current implementation the ports that have the enable/disable feature are used by the CIP (Complex Instruction Processor) processor and by the Peripheral Manager. The reason for this choice has to do with the fact that the nodes can only have one producer, that producer can only be the processor or a Peripheral. The other two access ports are connected to the Log Manager and to the Inter Module Communication Manager. The Log Manager needs full access to the memory because it is used to change addresss values via the outside world (USB), the Inter Module Communication Manager needs the full access to propagate the changes made to the values in the producer module to the modules where it is needed. This approach, also allowed saving FPGA resources, since there is no need for logic for two of the ports.

3.3.4 Log Manager

The Log Manager is responsible by the logging and debugging nodes stored in the GIMy of each module. The internal organization of the Log Manager is illustrated in Figure 15. The *Log Manager* allows the users to monitor the progress of the values generated by the FDEF processor or by the Peripherals. This monitoring can be done in two ways, live monitoring and mass storage logging.

The live monitoring is used to debug the FDEFs in the IDMS, since the values can be displayed directly in the graphical part of the functions. The live monitoring is implemented in C#, it sends read request to the GIMy address corresponding to the node we want to know the value. The read request is received by the *Log Manager* (Figure 15) and processed in the *Log Configuration* state machine. After processing the command, the state machine sends a response via USB to the IDMS, which then signals the graphical interface that a new value was received; the value is then presented to the user in the various forms that the IDMS offers. This debugging technique has some limitations, in terms of bandwidth because the messages need to be transferred via USB to the host computer and also the C# limitation in terms of the timers used to send the messages, that can be activated at a maximum frequency of 16ms. This functionality will be explained better in the software chapter.

In some cases, the user will want to monitor the values of a certain node during a race. Since the ECU will not always be connected to a PC, it has to have some internal mass storage; this is done using a MMC card. The block *Log Value Fetch Unit* illustrated in Figure 15 reads from the *Log Table* the GIMy addresses that are defined by the user to be stored in the mass storage. The enable/disable of the log is set in the IDMS and downloaded to the *Log Table* via a configuration message.

Figure 14 shows the format of a *Log Table* entry. Each entry contains a bit enabling or disabling the log and a GIMy address in case the log is active. When the log is not active the address field is not taken into account.

The *Log Manager* block is also used to configure the write enable tables stored inside the *GIMy* block.

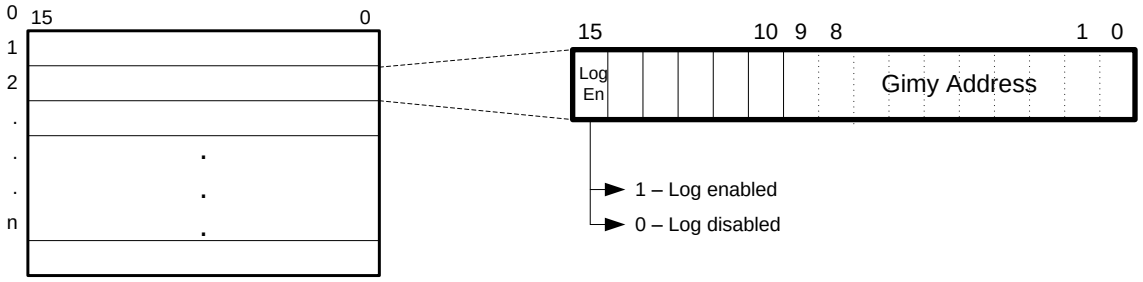

Figure 14 - Log Table entry format

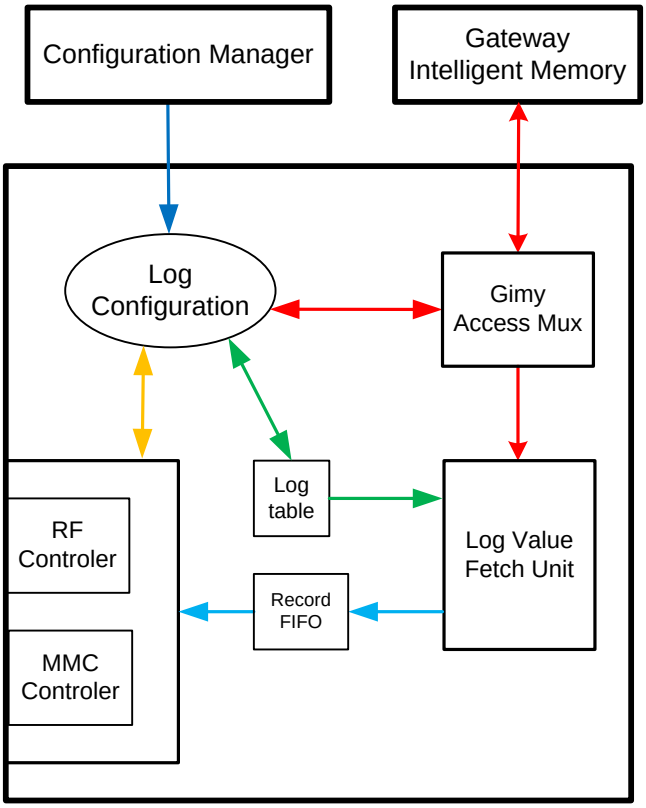

Figure 15 - Log Manager

3.3.5 Peripheral Manager

As said before, the sensors and actuators communicate with the modules via a Peripheral Bus. Each device (ECU modules and peripherals) connected to the peripheral bus, have a Peripheral Manager that implements the communication between the devices connected in the bus. Each Peripheral Bus can connect up to 16 devices, one of the devices must be a module set as bus master, so that means that we can communicate with up to 15 Peripherals with one Peripheral Bus. That communication is done via the Peripheral Manager of each Module.

Each Peripheral has a unique plug Id when connected in the bus. The plug Id takes values between 1 and 15, the value zero is reserved to the ECU Module, which is the bus master.

In Figure 16 the *Peripheral Manager* structure is presented. There are 4 state machines:

- **Message Controller**: This state machine receives and processes the messages that are sent to the *Peripheral Manager* via the *Configuration Manager*. There are several types of messages decoded by this state machine, it can be messages to update the internal tables, *Gimy2Per* or *Per2Gimy*, to exchange data to the Peripheral via the *DataFlow Controler*,

enable or disable the Peripheral communications or set the Peripheral controller as bus master.

- **DataFlow Controller**: This state machine implements the communication done in the Peripheral bus. It receives data from the *Peripheral Controller*, from the *Message Controller* and exchanges GIMy values based on the configuration set in the *Gimy2Per* table.
- **Message Up**: This state machine is responsible for formatting the message that is received from the *DataFlow Controller* so it can be sent via USB to the IDMS. The data is transferred from the *DataFlow Controller* to the *Message Up* using a FIFO.
- **MUX:** This state machine implements a multiplexer to allow the *Message Controller* and the *DataFlow Controller* to communicate with the *Configuration Controller* using the same channel.

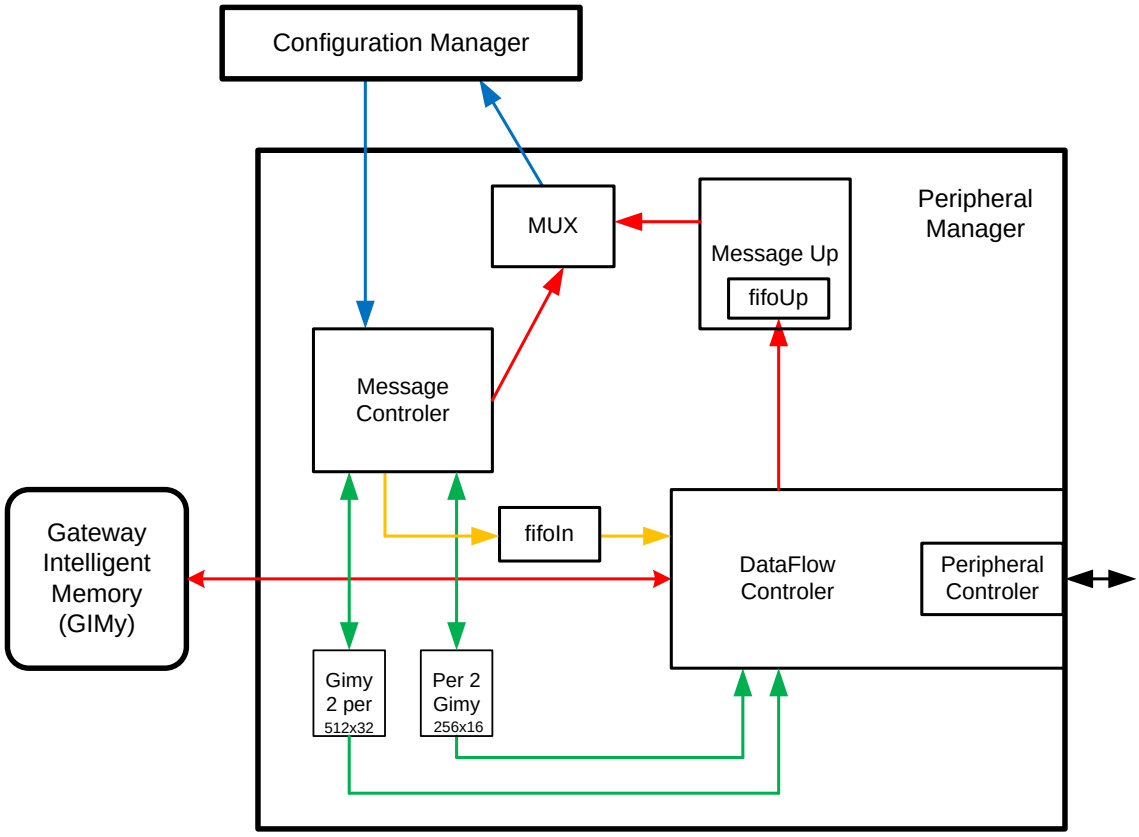

Figure 16 - Peripheral Manager

In normal operation, the exchange of node values between the Module and the Peripherals is always initialized by the Module. To exchange the node values the ECU Module sends two types of messages to the Peripherals:

- Write to GIMy : Message to write in the GIMy memory of the peripheral. This message contains 3 fields: plug Id, message Id and the value to write in the Peripheral. The plug Id is the number of the plug where the peripheral we want to communicate with is connected, the message Id is the id of the node we want to write in the Peripheral GIMy, there is a limit of 16 values we can exchange with

each peripheral, this limitation exists because the message Id has only 4 bits. The Peripheral after receiving this message will use the table *Per2Gimy* to retrieve the GIMy address to write the value received based on the plug Id and the message Id.

- Read from GIMy request : Message to read from the GIMy memory of the Peripheral. This message contains 2 fields: plug Id and message Id. The module sends this message to request a read from the Peripherals. The Peripheral then uses the plug Id and message Id to retrieve from its internal *Per2Gimy* table the GIMy address that is must read, then reads the value from GIMy and sends it back to the Module. After receiving the response message, the ECU module, uses again the plug Id and the message Id, to retrieve from its *Per2Gimy* table the address to write the received data.

The Peripherals only send messages in response to the read from GIMy request messages sent by the Module.

The node routing in the Peripheral bus is done using 2 tables. The tables are:

- **Gimy2Per**: This table that is shown in Figure 17 is used to store the commands that the *Module* must send to the *Peripherals*, either to send data from the *Module* to the peripherals with the write command or to request a value from a *Peripheral* sending a read from GIMy request command.
- **Per2Gimy** : This table is shown in Figure 18, and is used to translate the plug id and Message id of the messages used to read or write nodes from GIMy to the correspondent GIMy address. The address used to access the table is the concatenation of the plug id and the message id. For example, the message with plug id 0x5 and message id 0x3 will access the address 0x53 of the Per2Gimy table.

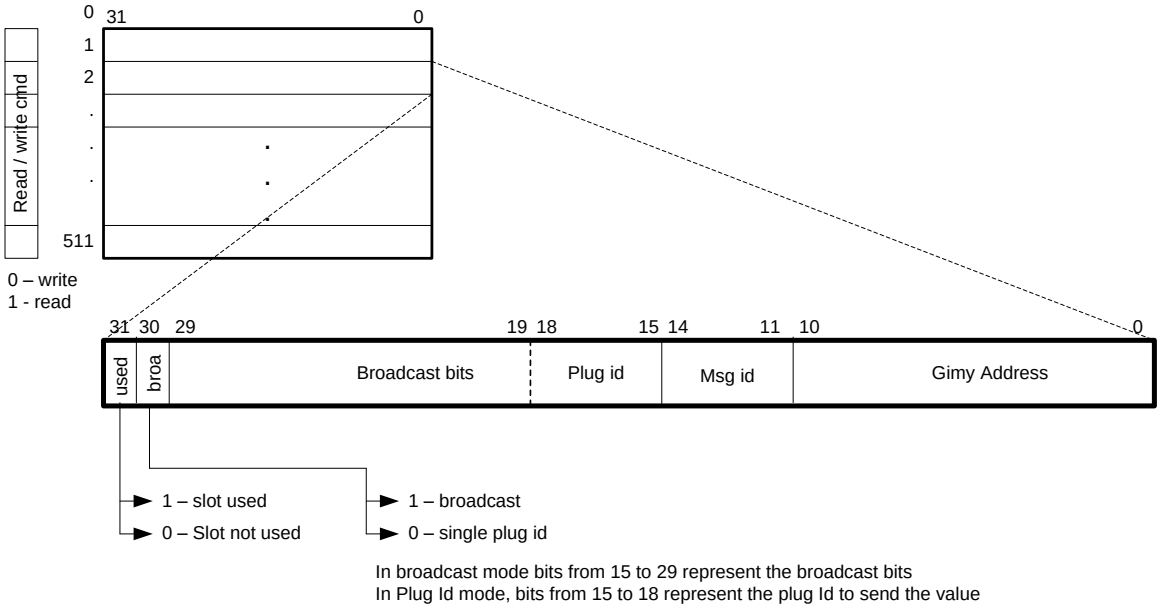

Figure 17 – Gimy2Per Table

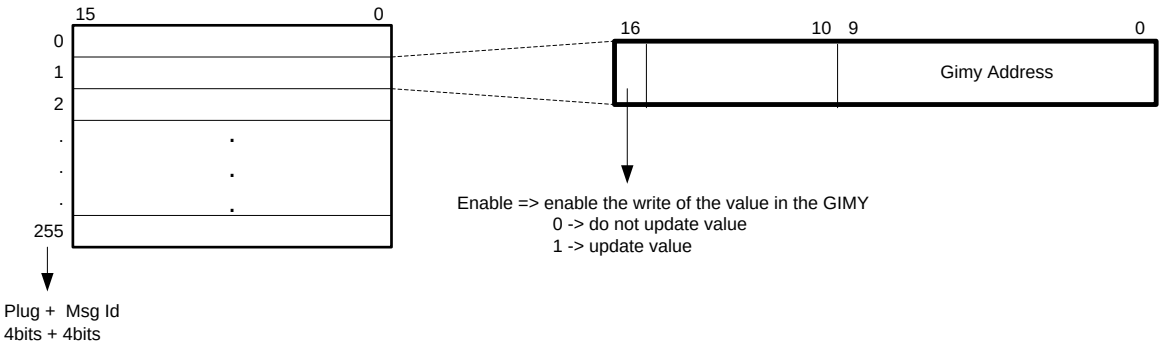

Figure 18 - Per2GimyTable

3.4 Support Software and API for the graphical interface

In this work it was also developed a set of functions that allow the graphical interface to interact and configure the hardware. These functions implement several functionalities, some of them are:

- Allocation of the Functions in the modules.
- Distribution of the Peripherals
- Generation of the routing tables
- Programming the ECU and the Peripheral
- Topology finder
- Serial communication detector
- Node generator
- Live debug
- Project Lists

Before explaining the previous enumerated functionalities, there are some basic principles that need to be explained, to allow a more easily understanding of the rest of the chapter.

Some basic principles were defined to allow an easier to code automatic allocation process. The principles are:

- A function is a group of Complex Instructions (CI)
- The group of CI's that form a function cannot be separated to different modules. (i.e, all the CI's of the same function must execute in the same module).
- Nodes/variables can only be changed by one entity. This entity can be either a function or peripheral. The value of the node is propagated from the producer to the consumers. The Node can have multiple consumers.
- Each Peripheral Bus can have a maximum of 15 peripherals

One of the objectives of this work was to provide the user with an automatic system that would be capable of distributing the functions and peripherals, calculate the routing tables, generation of the Complex Instruction code and program all the peripherals and the modules that make the ECU.

This could be divided in two parts; the first one was called *Allocation* and includes the distribution of the functions and peripherals, the calculation of the routing tables and the generation of the Complex Instruction code. The second one, called *Deploy* includes the programming of the routing tables and the complex code in the peripherals and in the modules of the ECU.

The allocation process is implemented in separate C# classes for the ECU modules and peripherals, the implementation for the ECU modules is done in the class *Allocator.cs*, for the peripherals is implemented in the class *PeripheralAllocator.cs*. This separation is done

because there are some issues that are different for the allocation on ECU modules or Peripherals. Some of them are the fact that the functions in the peripherals are only assigned to the Peripheral instead of different modules.

In both classes that implement the allocation process, function *ProcessAllocation()* calls the several functions that implement the necessary steps to perform the allocation.

The allocation process was separated in several functions to allow an easier change and simplify the allocation algorithm. In the following chapters the different steps of the allocation process are explained. The function *ProcessAllocation()* starts by checking all the references to the project, the ECU description and the several lists of objects, such as functions, nodes and peripherals. If any error occurs it is reported to the graphical interface.

The two main steps of the allocation process are the distribution of the functions and the distribution of the peripherals among the modules. The functions that implement these two steps are called first. Then we calculate the routing of the nodes in the modules; calculate the routing tables for the modules, the routing tables for the peripherals and the logging tables. After that we call the function that create the binary code that is going to be executed in each module, and finally we call the function that generate the data that to configure each module and that is going to be stored in the flash memory of the FPGAs boards.
So now a brief explanation of some of the steps described above is presented.

3.4.1 Allocation of the Functions in the modules

As said before, the first step in the allocation process is to distribute the functions in the modules, this distribution is automatic, but there is also the option for the user to specify the location of any function. The user can set a function to be executed in any of the modules that make up the ECU.
Due to lack of time to get a prototype of the whole system, the allocation of the function is done manually by the programmer. The functions are distributed in terms of number of functions in the modules. This is, if we have 10 functions and 3 modules, one module will get 4 functions and 2 will get 3 functions.
Some code was made to implement an algorithm that would take into account some rules. Some of the rules that where going to be taken into accounts are:

- The size of the functions and the corresponding predicted execution time.
- The numbers of nodes the function shares with other functions, and where are those other functions allocated.
- The number of nodes the function shares with peripherals and where are the peripherals located.
- The modules current load, this is, take into account the remaining processing capability of the modules after allocating several functions.
- The module capability to allocate the nodes, since we have a limited number of nodes in each module, this limit is due to the memory available inside the FPGA.

3.4.2 Distribution of the Peripherals

The distribution of the peripherals is automatic, but the user, like in the case of the functions, has the ability to set the location of the modules, the location setting of the module by the user can have three variations:

- Module lock: The user specifies the module in which the peripheral must be located, but the plug choice is left to the allocation algorithm. This is used when the user want a peripheral to be located in a specific module.
- Plug lock: The user specifies the plug number in which the peripheral must be connected in the peripheral bus of the module, but the module is the allocation algorithm that decides. This can be used when the user want a specific peripheral to be assigned to a specific plug id.
- Module and Plug lock: In this option the user specifies the module and the plug for the module to be inserted.

The distribution of the peripherals was another interesting topic that had to be left behind due to the lack of time. In the current implementation we start by checking all the peripherals to see if they are allocated by the user and in that case if they have valid locations. Then the algorithm starts by checking the peripherals that are not assigned and sets their location to the modules that have less peripherals connected. This way in the end we must have a equal distribution of the peripherals in the modules.

As in the case of the function distribution, some work was made to improve this algorithm. A priority field was added to the description of each peripheral, this field would vary between 0 and 100, being 100 the more priority one. This would mean that a peripheral with priority 100 would require more communications than a 0 priority one.

This would allow the algorithm to separate the high priority ones in different modules to allow a larger bandwidth in the communication with those peripherals. Or we could put all the priority 0 peripherals in the same bus, because the communication deadlines for their data could still being achieved.

3.4.3 Programming the ECU and the Peripheral

The programming of the ECU and Peripheral is done by writing the internal configuration tables, the complete configuration messages and the code instructions to the external memory. The configuration messages are sent to each module block responsible for the table we want to program. Since the configuration tables are stored in ram, there values are lost after a power down, so to restore the tables with the correct values we write the configuration messages to the external memory of the module. When the module power is turned on, the configuration messages are fetched from the external memory by the Configuration Manager that then distributes the messages as if they were sent via the USB interface.

3.4.4 Topology finder

The topology finder searches in the connected devices for either an ECU Modules or Peripheral Module. After detecting which ones are ECU Modules, it sends different values to a specific GIMy address of each module; those values are then automatically transmitted to the modules that are on the left and right of each other. After this, we read those addresses from all the modules and have the values of the modules that are on the left and on the right of each module. This allows the IDMS to determine the topology of the ECU. In Figure 19 the demonstration of this method is presented using the example of an ECU that has 3 modules. The first step is to write to the top address of each module a different value. In this case we write the value A, B and C to the modules 1, 2 and 3 respectively. After writing the values they are transmitted by each module to the right and left modules. After this we read the 2 address's that contain the module that are on the left and right.

We use two addresses to determine the ring topology, so we can detect more precisely which is the module that is not communicating correctly.

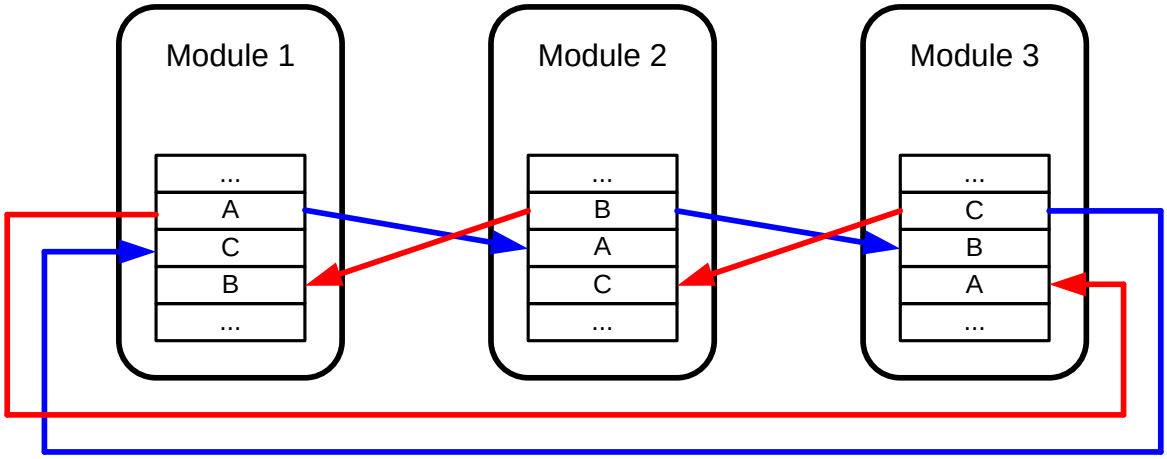

Figure 19 - Topology finder

3.4.5 Serial communication detector

To allow the user to check if the serial links are active, the *Serial Communication Detector* was implemented; its functionality allows the detection of a broken serial link between two adjacent modules.

To allow this detection the receiving state machine on the master side of each module has 3 registers:

- *Received Messages*: Number of corrected messages received.
- *CRC Errors*: Number of messages received with errors in the CRC.
- *Start Errors*: Number of message that has an error in the start byte.

Associated with each register there is a timestamp register that is written every time the associated register is also written.

The *Serial Communication Detector* reads these 3 registers and based on that information can determine if the serial link is active or broken.The C# class *SerialCommStateDetecter.cs* implements this functionality. The class supplies the following functions to the user:

- SerialCommStateDetecter() : This is the class constructor that initializes all the necessary variables, including the timer that is going to be used to scan the registers, which as to be started with the next function.

- void StartSerialCommStateDetecter(double milliseconds_p) : This functions starts the Serial Communication detector. The parameter milliseconds, is the time in milliseconds that will set the period used to scan the registers in each module; this value is limited to 15 milliseconds by C# and the .Net platform.

- void StopSerialCommStateDetecter() : This functions stops the serial communication state detection.

There are two functions on this class that are hidden from the user and that actually do the work of detecting a broken serial link, those functions are:

- void serialCommStateDetecterTimer_Elapsed(object sender, ElapsedEventArgs e) : This function is executed each time the timer interval elapses. Each time it is executed it sends a read request of the Received Messages register of every module in the ECU.

- void responseMessageReceived(MessageAckTimeoutConfig sender, EcuCommand receivedCommand) : This message is executed each time a response arrives, after the request made in the previous function. The response message has the value of the Received Messages register and the timestamp it was last written. The received register value and the timestamp are compared to the previous received one, and if they are the same, a broken link is detected. If they are different, the serial link is working. If the state of the serial link is changed, either from broken to active or active to broken, the S*erialCommsStateChanged* event is launched. This event signals the graphical interface that the serial communication state has changed, passing the ECU Module as argument.

In the current implementation only the register *Received Messages* is used to detect the broken serial link, in future implementations the other two registers: *CRC Errors* and *Start Errors* would be also used. Because there are some situations that an error can occur but we do not detect it with the current software implementation. The way to implement it also using these two registers would be done in the same way, we would make a read request to them, and then compare the response with the previous received data, and detect the errors. This was not implemented do to the lack of time.

3.4.6 Node Generator

The *Node Generator* allows the user to set values to every variable in the system. This is useful when debugging the motronic functions in the hardware. This way the user can set a value to a particular variable and watch its influence on the system.

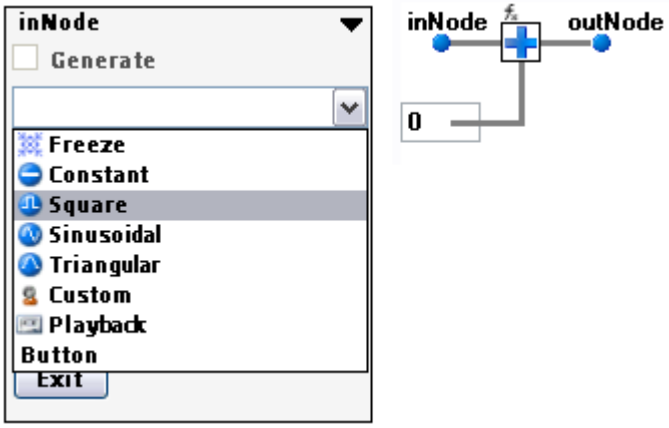

Figure 20 - Node Generator in function view

To use the *Node Generator*, the user just has to click with the right mouse button on the variable he/she wants to generate values for. After clicking the user chooses the *Generate* option and a dialog box like the one shown on Figure 20 will appear. In this dialog box the user sets the type of generator he/she wants to use, and its parameters.
Currently there are 6 generators implemented, they are:

- Freeze: This generator disables the writing of new values in the GIMy memory address associated with the variable we want to freeze, that is, the GIMy internal mechanism will discard any update requests on that particular address.

- Constant: This generator sets a value and disables further writings in the GIMy memory address associated with the variable we want to change.

- Sinusoidal: This generator disables writings in the GIMy memory address associated with the variable we want to generate, and then sends periodically values to that address, simulating a sinusoidal wave. Currently the user can set the frequency, the amplitude and the offset of the sinusoidal wave. The calculation of the sinusoidal value is done with the following expression.
  value = amplitude * Sin(2 * Pi * freq * time) + offset;
  were the value *time* is calculated using the Ticks counter supplied by the computer, when we start the generator, the current Tick value is stored, and every time we request a new sinusoidal value, the time elapsed is calculated and the new sinusoidal value is sent to the GIMy address.

- Triangular: This generator disables the writings in GIMy memory address associated with the variable we want to generate, and then sends periodically values to that addresses, simulating a triangular signal. The user can set the frequency and the amplitude of the triangular signal.

- Custom: The custom generator was not implemented, but as the name suggests, it was going to be a generator that would have an expression as input, and would have a mechanism to evaluate the expression and send the result to the node.

- Playback: The playback was not implemented, but it would allow the generation of values based on a previously recorded log of a node. This way we could inject the values recorded during a race and simulate it. This can be used to replay the values generated during a race and find errors that otherwise would not be detected.

- Button : This is a on/off button generator. When selected a button appears in the dialog box. When the user presses the button, the value "1" is sent to the GIMy address of the node, when the button is released a "0" is sent to the GIMy address of the node.

3.4.7 Live debug

The live debug is used to supply the user with the ability to monitor the values of the nodes in the function view and in the project list view. This is a request based reading, for each node we want to debug, the IDMS sends a message with a read request to the module were the node is being generated, and the module responds with the value of the node. This functionality is implemented in the class *NodeLiveLog,*in file NodeLiveLog.cs. This class has a list of the nodes that have the Live debug functionality activated. The class provides several methods:

- void StartLiveLog(double milliseconds_p) : This function starts the Live debug. The argument millisecond_p is used to program the time of the timer used to trigger the function that sends the message requests to the modules.

- void StopLiveLog() : This function stops the Live Debug timer.

- void ResetLiveLog() : This function stops the timer and removes all the nodes from the Live debug list.

- void EnableTempDebug(NodeEdit nodeEdit_p, TNode node_p) : This function activates a temporary Live debug for the TNode passed as argument. This is used to provide Live debug to the current selected node in the Graphical Interface. It provides a means to update the node that the user is currently selecting without adding it to the normal Live debug list.

- void DisableTempDebug(TNode node_p) : This function disable the Live debug for the TNode passed as argument.

- void AddNodeToLiveList(TNode node_p) : This function adds the TNode passed as argument to the Live debug list of nodes that must be debugged.

- void RemoveNodeFromLiveList(TNode node_p) : This function removes the TNode passed as argument from the Live debug list.

- void nodeLiveLogTimer_Elapsed(object sender, ElapsedEventArgs e) : This is the function that is called when the timer elapses. This function sends a read request for each node that is in the Live debug list.

- void responseMessageReceived(MessageAckTimeoutConfig sender, EcuCommand receivedCommand) : This function is executed every time a message response to a read request is received. The information we receive is the serial number of the module that responded, the address of the GIMy memory and the data associated with that address. So first we extract the moduleId in the ring, and then we search in the module list that we have locally for that particular module, and then we get the nodeId associated with the address in the GIMy memory from the local module information. We then update the value of the node, and send an event to the graphical interface to inform that a new value is available.

3.4.8 Project lists

The Project Lists, is the name given to a tab in the IDMS that allows the user to access the all the functions, nodes, Peripherals and modules in a list format. This can be used to generate reports of

There are 4 different views in the Project Lists tab:

- Nodes
- Modules
- Functions
- Peripherals

Next each of these different views will be explained.

3.4.8.1 Nodes tab

The nodes tab is show in Figure 21. In the left menu we have a list with all the nodes that exist in all the functions, including the ones that are internal to the peripherals. Each node has a context menu associated with it. In that context menu, we have access to several options; those options are presented in Figure 22.

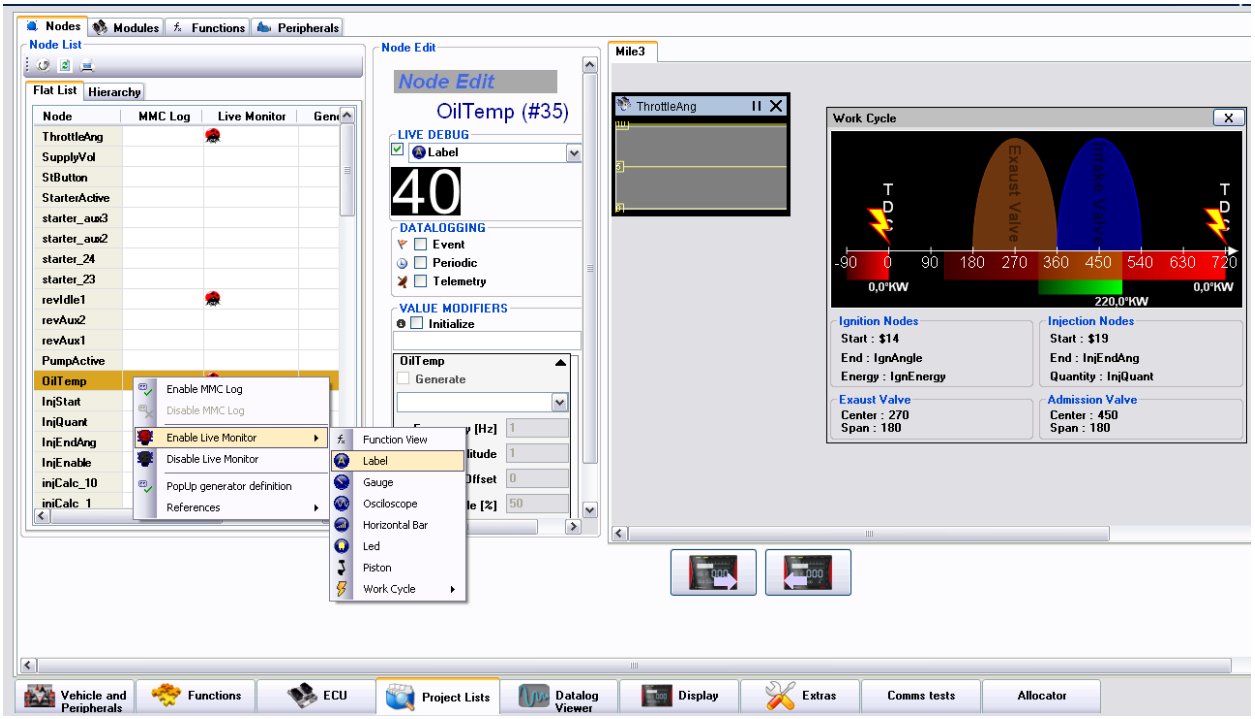

Figure 21 - Nodes tab overview

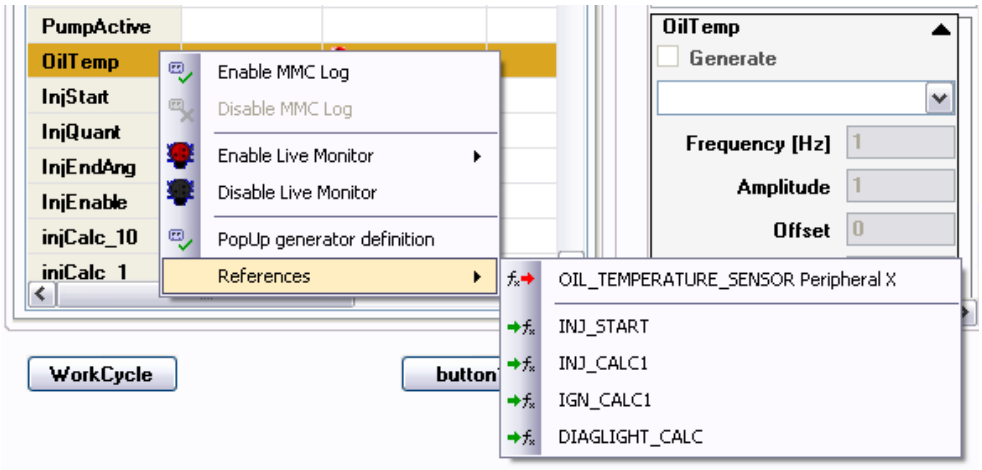

Figure 22 - Node tab context menu

3.4.8.2 Module tab

The Module tab contains a list of all the modules that make up the ECU. For each module the following information is shown:

- Number of functions
- Number of instructions
- Cycle time for the execution of all the instructions
- Number of peripherals connected
- GIMY occupancy
- Data log occupancy
- Master Serial Communication cycle time

- Slave Serial Communication cycle time

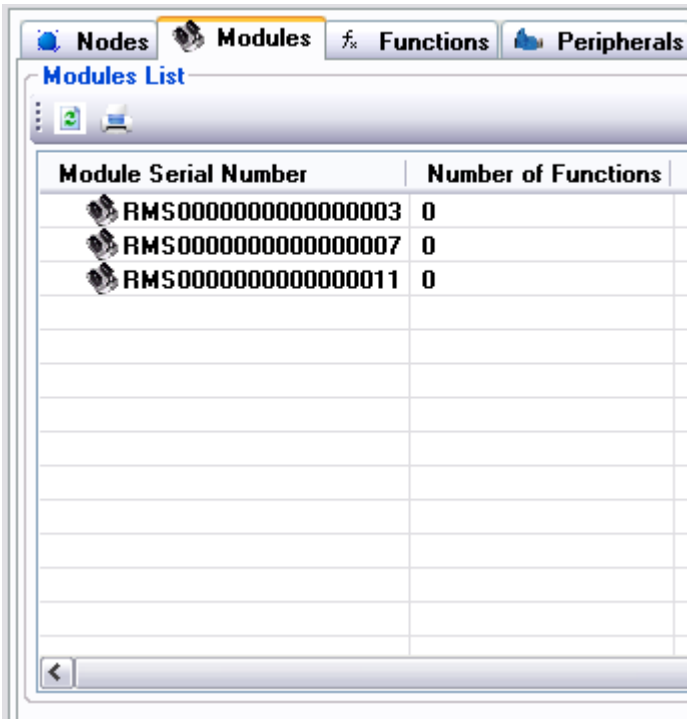

Figure 23 - Module Tab

3.4.8.3 Functions list tab

The Function tab shows all the functions used in the project; This tab contains 2 lists: a functions list (Figure 24) and a instruction list (Figure 25). When you select a function from the function list, the instructions of that function are shown in the instruction list.

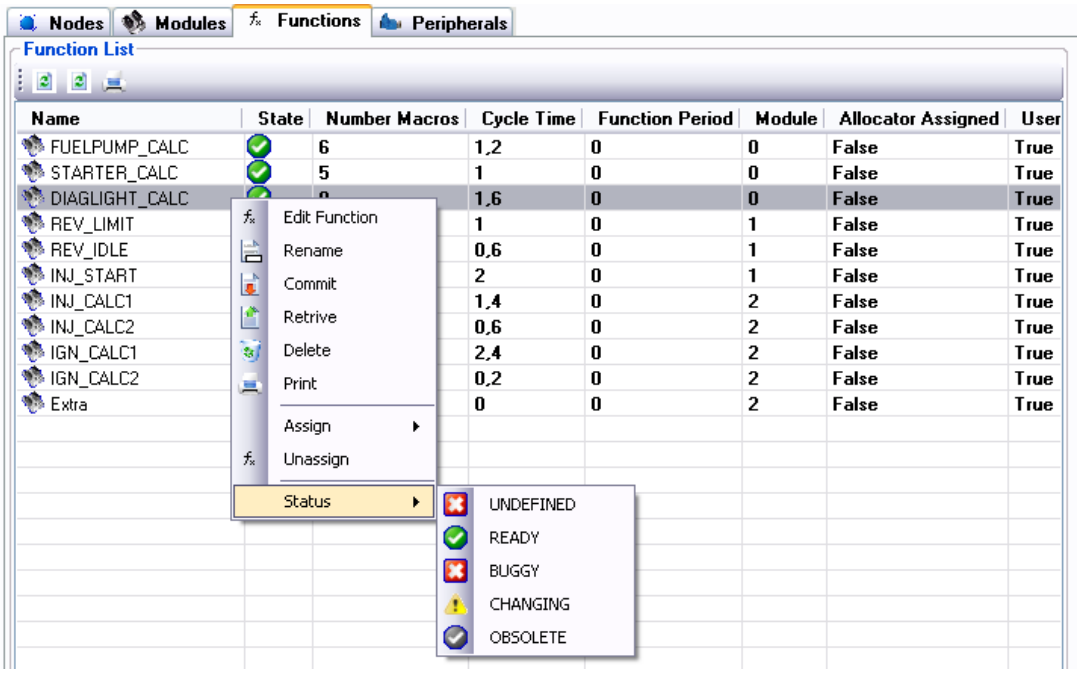

Figure 24 - Functions List tab

Macros from function : DIAGLIGHT_CALC

Index	Name	Inputs	Outputs	Path	Prio	Secret (0)	Custom (0)
1	DLYA	EngStarted 50000	diag1	fx	ALWAYS		
3	COMS	FuelPressure 3	diag2	fx	ALWAYS		
4	NOT	diag1	$18	fx	ALWAYS		
5	OR	$18 diag2	diag3	fx	ALWAYS		
7	COMS	SupplyVol 11	diag4	fx	ALWAYS		
8	OR	diag3 diag4	diag5	fx	ALWAYS		
10	COMB	OilTemp 90	diag6	fx	ALWAYS		
11	OR	diag5 diag6	diag7	fx	ALWAYS		

Figure 25 - Instructions List

3.4.8.4 Peripheral tab

The peripheral tab shows all the peripherals that are declared in the project. Figure 26 shows a example of a list of peripherals. This list allows us to perform several actions on the peripherals by accessing a context menu.

The actions that can be made are:

- Implement: This converts the functions of the peripherals to CIP language, and generates all the necessary routing tables for the peripheral.
- Deploy: Transfers the code and tables generated to the peripheral.
- Assign Plug : Used to assign the peripheral to a specific module and plug in the system.
- Assign USB: This allow the user to set the peripheral to a connected USB device, allowing the IDMS to communicate with the peripheral via USB. In a later phase we would like this to be an automatic process. The IDMS should be able to recognize that the USB device is a peripheral and which one it is. The Peripheral also can be connected via USB to program and debug when necessary.

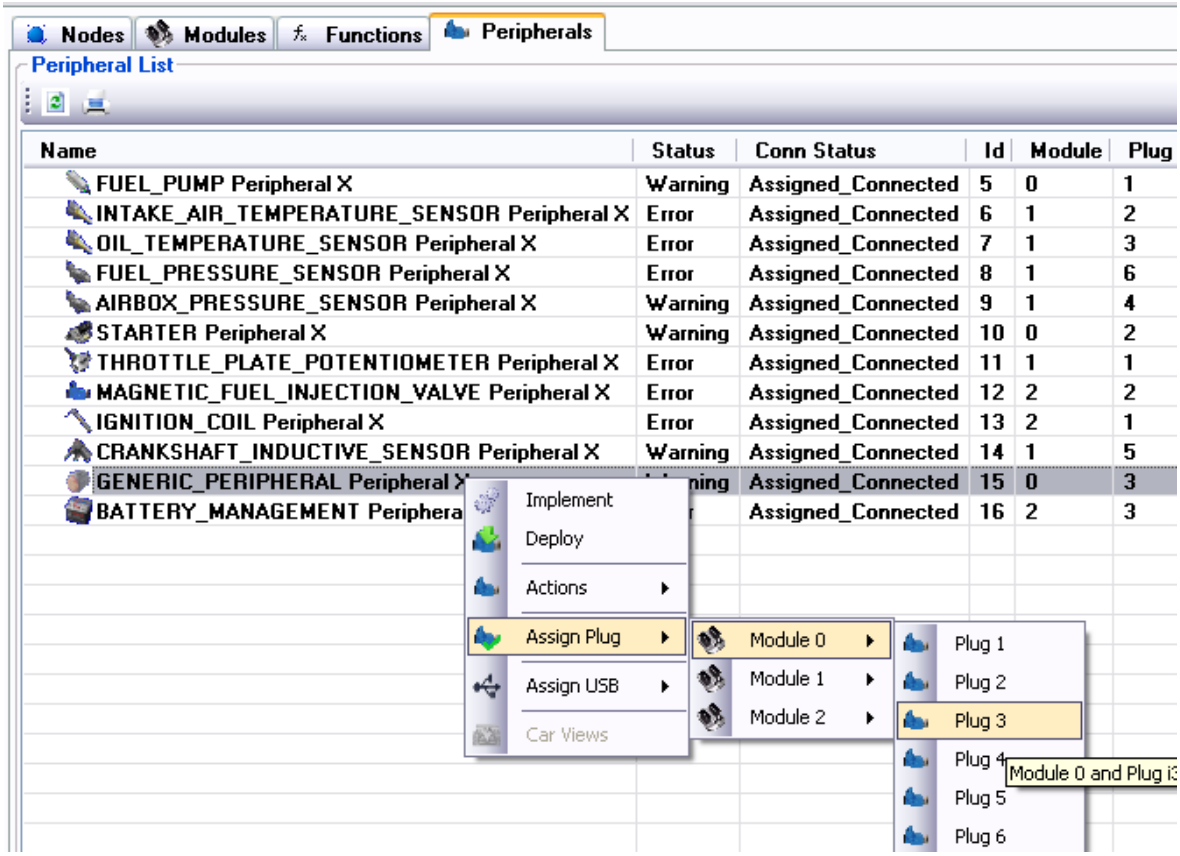

Figure 26 - Peripheral Tab

3.4.9 Display tab

The Display is the name given to a tab in the IDMS that allows the user to set live monitoring to project nodes, and place them in the screen as if it was a traditional dashboard. The user can create multiple dashboards and go through all of them, to he/she presses the arrows buttons in the bottom of the screen. One of the main uses of this tab would be the creation of custom dashboards, depending on the user choice.

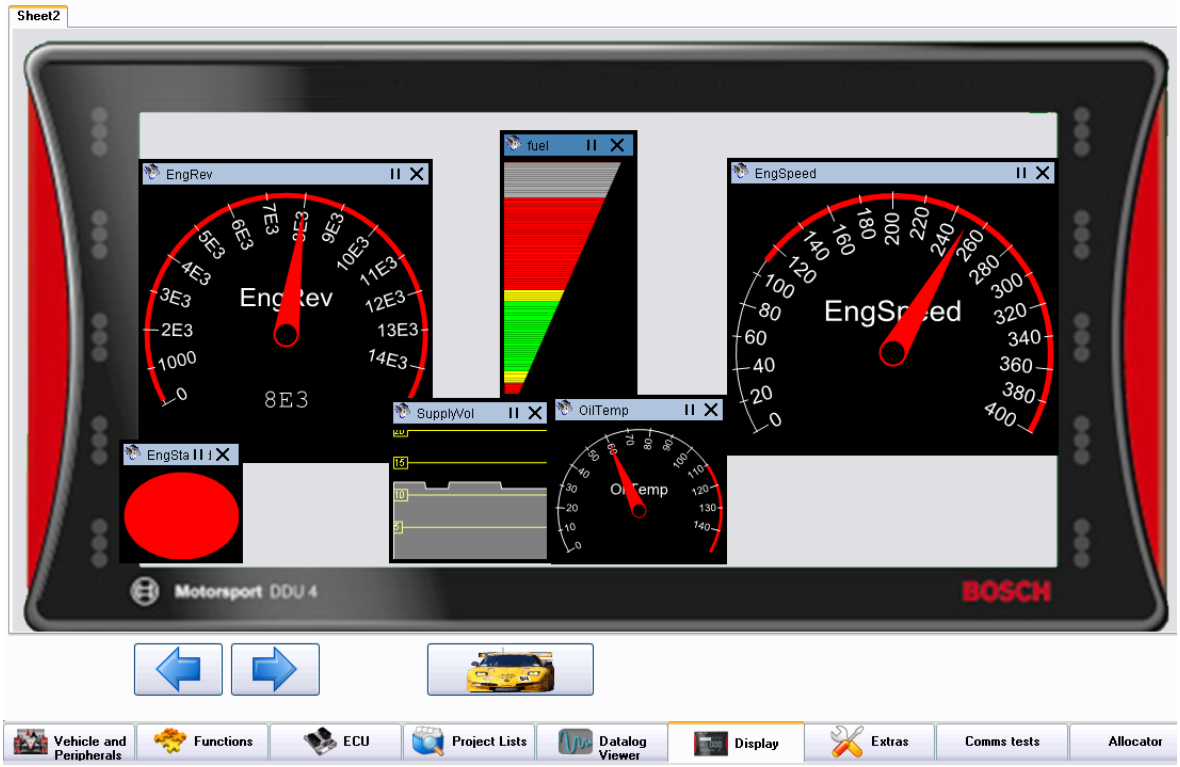

Figure 27 - Display Tab

Chapter 4
Results

4.1 Summary

In this chapter the results obtained are presented. The result of the module implementation and finally a running prototype controlling a one cylinder engine is presented.

4.2 Module implementation

As said before the module was implemented in a FPGA. Since the module had several blocks, an integration process of all the blocks had to be made. Do to the definition of the communication signals of the different blocks in the beginning of the project; the integration of the blocks in the module was achieved with minor changes and in a relative small amount of time. After the integration of the blocks, each block was tested and confirmed that it was working correctly. The FPGA utilization summary is shown in Figure 28.

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Total Number Slice Registers	9,711	29,504	32%
Number used as Flip Flops	9,506		
Number used as Latches	205		
Number of 4 input LUTs	22,168	29,504	75%
Logic Distribution			
Number of occupied Slices	13,878	14,752	94%
Number of Slices containing only related logic	13,878	13,878	100%
Number of Slices containing unrelated logic	0	13,878	0%
Total Number of 4 input LUTs	24,827	29,504	84%
Number used as logic	22,168		
Number used as a route-thru	1,075		
Number used for Dual Port RAMs	1,408		
Number used as 16x1 RAMs	176		
Number of bonded IOBs	114	250	45%
IOB Flip Flops	14		
IOB Latches	1		
Number of Block RAMs	22	36	61%
Number of GCLKs	11	24	45%
Number of DCMs	3	8	37%
Number of MULT18X18SIOs	1	36	2%
Total equivalent gate count for design	1,805,615		
Additional JTAG gate count for IOBs	5,472		

Figure 28 - Module FPGA Utilization Summary

4.3 Running Prototype

A running prototype was built using the ECU2010 concept; this prototype controlled a mono-cylinder engine with the following characteristics:

- **Brand:** *Honda*
- **Application:** lawn-mower
- **Displacement:** *25 cm^3*
- **Type:** 4-stroke
- **Fuel:** lead-free gasoline

The final demonstration of the prototype took place on November 28th, 2008. The engines as well as the ECU were on the University of Aveiro, but the demonstration was shown at Bosch Motorsport headquarters in Markgröningen, Germany. This was due to another functionality implemented in the IDMS, the *"Tele-Operation",* which allowed the developer to interact with the hardware platform using the internet as if it was right next to it. During the demonstration all the functionalities were demonstrated and worked with no failures. The dataloging extract was not shown, due to bandwidths constraints.

The hardware platform used in the prototype was composed by 3 modules for the ECU (Figure 30), and 12 peripherals (Figure 31). This platform allowed the demonstration of the processing of instructions in every module, communication with sensor and actuators, communication between modules with the transfer of internal nodes and peripherals nodes.

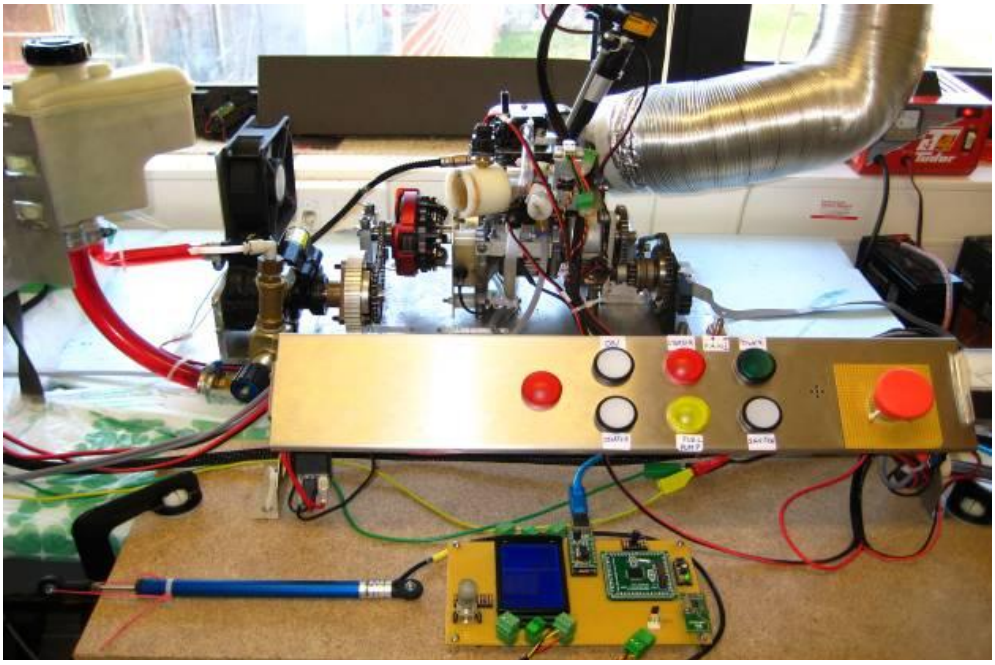

Figure 29 - Mono-cylinder engine testing platform

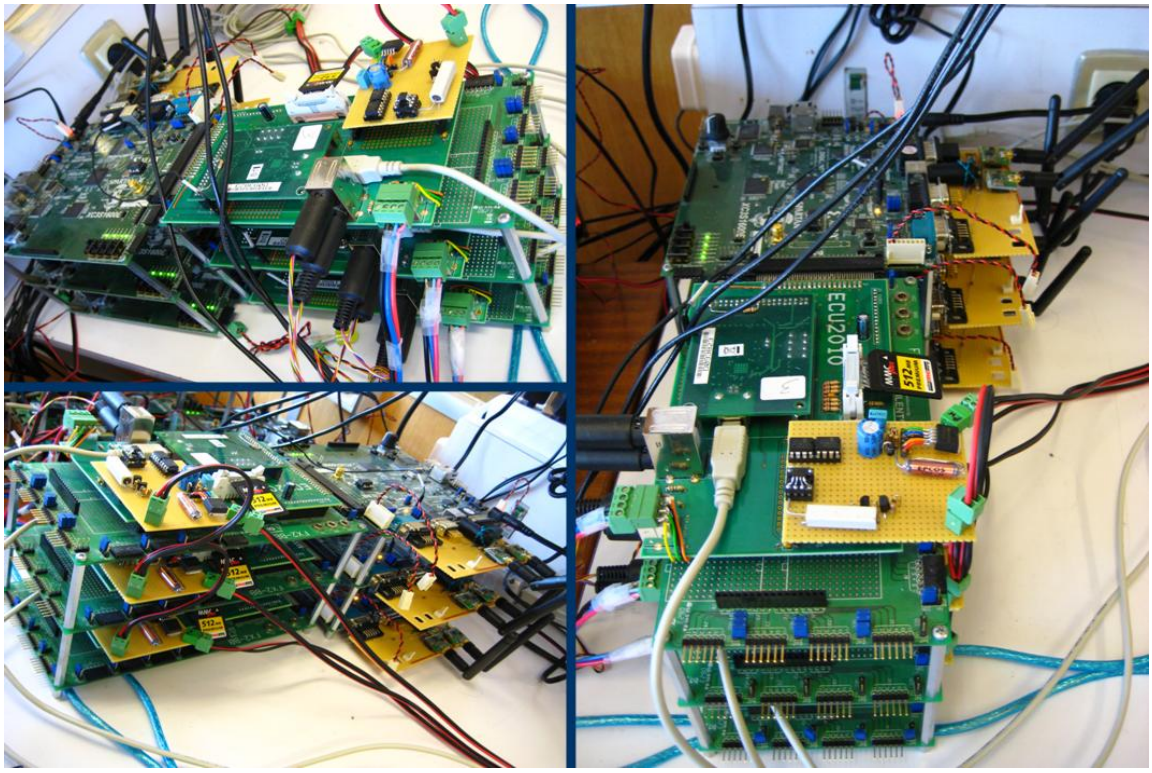

Figure 30 - ECU from different views (3 modules)

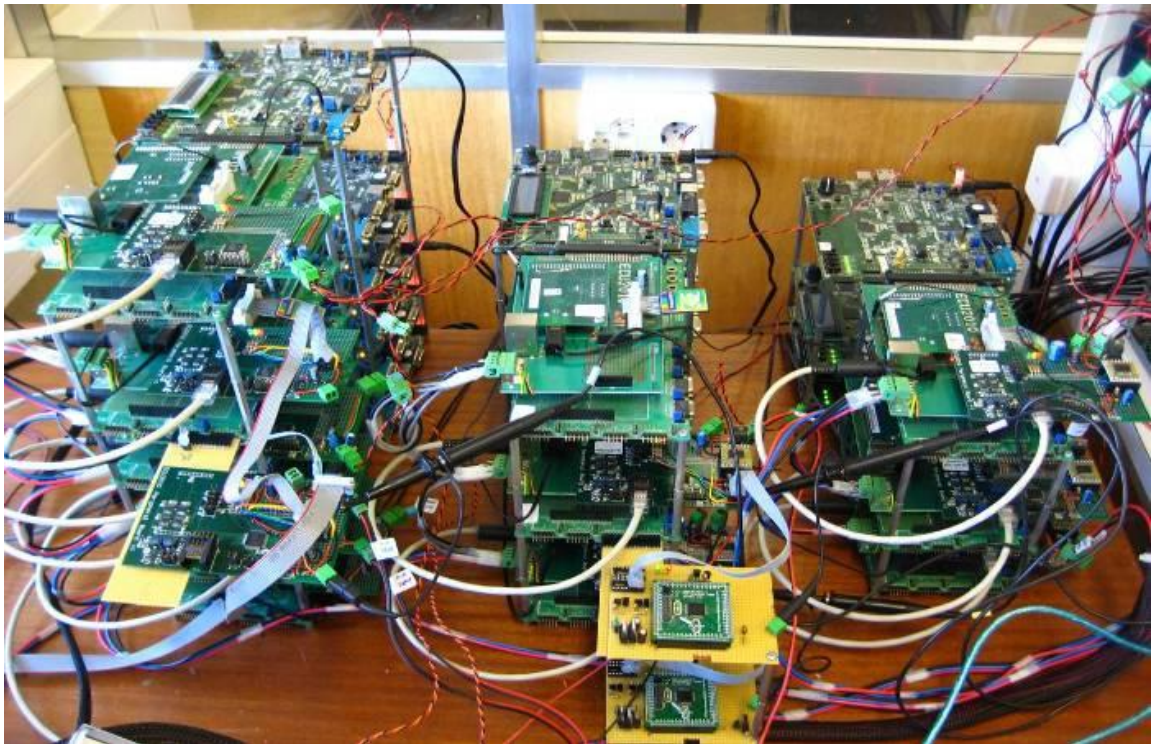

Figure 31 - Peripherals (Sensors and actuators)

Chapter 5
Conclusions

5.1 Summary

Due to time constraints of the ECU2010 project and the objective of proof-of-concept being of the most importance, some of the project implementations options could have been different, in this chapter some improvements and future work is presented.

5.2 Improvements and Future work

After the final implementation of the module, the FPGA usage was almost total. After some research and some tests, the team realized that a different approach for the development of each block inside the module could have been taken. In the current implementation all the functionalities of the blocks were implemented using only logic blocks, another option could be the use of a synthesizable microprocessor in each block to perform some of the functionalities. There are several microprocessor available, one of the possible choices would be the use of PicoBlaze, which is a free microprocessor supplied by XILINX that also supplies an IDE to program and debug the code running in the microcontroller. The use of microcontrollers would allow a faster development and test of different implementations and reduce the space used in the FPGA, which in return could lead to an overall speed increase because less signals would be needed to be routed in the FPGA.
Another aspect that could be improved and future work made is in the resource allocation in the parallel system, that is, the distribution of functions and peripherals in the modules. Do to the lack of time to present a functional prototype, this aspect had to be left behind, but it is my opinion that this part is one of the most important, since the automatic allocation done with optimized algorithms that would take into account the functions and peripherals importance and defining timings objectives for nodes actualizations would obtain a better distribution of resources and functions.

Glossary

BCDFP – Binary Coded Decimal Floating Point

CI – Complex Instruction

CRC - Cyclic Redundancy Check

C# - C Sharp

ECU - Electronic Control Unit

FIFO - First In First Out

FPGA - Field Programmable Gate Array

GIMy - Gateway Intelligent Memory

IDMS – Integrated Development and Management System

IP – Internet Protocol

MMC - Multimedia Card

SIMO – Slave In Master Out

SOMI – Slave Out Master In

SPI – Serial Peripheral Interface

TCP - Transmission Control Protocol

USB - Universal Serial Bus

VHDL - VHSIC Hardware Description Language

GIMy - Gateway Intelligent Memory

Bibliography

[1] dSpace GmbH, 2010, www.dspaceinc.com.

[2] ETAS GmbH, "ASCET Software Family", 2010, www.etas.com.

[3] Magneti-Marelli, "FastPRO ECU family", www.magnetimarelli.com, motorsport.magnetimarelli.com.

[4] XILINX, "FPGA chip XC3S1600E", 2010, www.xilinx.com.

[5] XILINX, "ISE Design Suite", 2010, www.xilinx.com.

[6] XILINX, "ChipScope Pro 12.1 Software and Cores – User Guide", UG029 (v12.1) April 19, 2010, www.xilinx.com.

[7] DIGILENT, "MicroBlaze Development Kit Spartan-3E 1600E Edition User Guide", UG257 (v1.1) December 5, 2007, www.digilentinc.com.

[8] XILINX, "Spartan-3E FPGA Family Datasheet", DS312 (v3.8) August 26, 2009.

[9] Microsoft Corporation, "Visual Studio 2005", 2010, www.microsoft.com.