Implementation of a traffic interceptor for Anybus® CompactCom™

FRANCISCO JAVIER MACOTELA LÓPEZ
JAVIER GAZTELUMENDI ARRIAGA
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Implementation of a traffic interceptor for Anybus® CompactCom ™

Francisco Javier Macotela López
fr6577ma-s@student.lu.se
Javier Gaztelumendi Arriaga
ja8602ga-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Erik Larsson

Company supervisors: Mattias Bornhager & Jonas Alowerson
(HMS Industrial Networks)

Examiner: Christian Nyberg

January 2, 2020

© 2020
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The aim of this project is to design and develop a traffic interceptor for the Anybus protocol. This protocol is owned by HMS Industrial networks and its use is intended to enable industrial devices to communicate with any fieldbus or industrial Ethernet network. The traffic interceptor developed in this master thesis was introduced so the host can observe the data flow before it is translated inside the Anybus products and more specifically the CompactCom product. It receives the signals from the communication between the Anybus and its host application, then these signals are formatted in an FPGA and processed in a microprocessor to finally be displayed using a protocol package analyzer called Wireshark.

Popular Science Summary

Modern industrial systems rely on real-time communications between machines, sensors, actuators, human control interfaces, etc. Traditionally, these devices are connected to each other using a fieldbus, an industrial communications network whose characteristics can vary depending on the requirements of the specific industrial process that is being controlled. Examples of requirements include bandwidth, latency, communication distance, fault tolerance, and many others. This has led to the existence of many different communication protocols, often sponsored by different vendors or groups of vendors, that are incompatible between them.

To add to this, the requirement for higher capacity and lower costs makes Ethernet an attractive value proposition for this communications, since the users could leverage existing hardware and software tools for TCP/IP communication at a much lower cost than the niche industrial protocols. However, Ethernet by itself has some drawbacks when it comes to industrial environments, such as lack of time accuracy, electrical noise sensitivity and diffculties to run for a very long distance.

To bridge the gap between Ethernet and the industrial communication requirements, several Industrial Ethernet variants have been created by adding extra functionality such as real-time control and determinism. This has resulted in an increase in the amount of industrial communication standards that exist, which are mostly incompatible between them. This increases the cost of setting up an industrial plant, since all the devices need to be compatible with each other, meaning that, once a protocol has been chosen, the user will be limited to those manufactured or compatible with the ones from the same company.

In order to solve this issue, HMS created the Anybus communication protocol. The goal of this protocol is to be an intermediate bridge between networks and devices that use different protocols. To do so, the traffic from each side is "translated" to the Anybus protocol, and from this common language, it is translated back to the corresponding network protocol. This allows for great flexibility in different scenarios: for example, direct protocol converters can be used to communicate a device with an uncompatible network. Another possibility is for a company to design a device that uses the Anybus protocol, so that the customer can connect it to different networks by changing the adapter.

While the Anybus protocol provides great flexibility, it is currently a blackbox

for the user. This means that the user has no way of monitoring the data that goes through the Anybus connection, so when an occasional error happens in any of the stages of the communication, or a wrong configuration is applied, it is difficult to troubleshoot and debug.

The goal of this project is to solve this problem by building an Anybus traffic interceptor that can be implemented with hardware that is already in use by HMS, allowing users to analyze the traffic through the Anybus by using standard tools such as an Ethernet card and Wireshark, an open source network auditing software widely used in the industry. The cost of implementation is greatly reduced by using existing hardware, avoiding the cost of a new electronic design and the reusability of existing FPGA modules.

Contents

1 Introduction 1

2 Related Work 5
2.1 Methods comparison . . . 6
2.2 Software and hardware implementation comparison . . . 8

3 Understanding Anybus CompactCom 11
3.1 Anybus CompactCom operation modes . . . 11
3.2 Physical Connectors . . . 12
3.3 SPI Pins . . . 13
3.4 SPI Interface Signals . . . 14

4 SPI Frame 19
4.1 SPI MOSI Frame . . . 21
4.2 SPI MISO Frame . . . 21

5 Traffic Interceptor Overview 23
5.1 General Overview . . . 23
5.2 PCB Description . . . 24
5.3 FPGA . . . 26

6 Traffic Interceptor Design 29
6.1 FPGA Logic . . . 30
6.2 AHB Bus . . . 30
6.3 Microcontroller Sub System (MSS) . . . 33
6.4 Ethernet . . . 33
6.5 Network Traffic Analyzer . . . 34
6.6 Software Oriented design . . . 34
6.7 Network Packet Analyzer Oriented design . . . 35
6.8 Logic Oriented design . . . 35

7 Traffic Interceptor Implementation 37
7.1 Traffic interceptor wrapper with MSS . . . 38
7.2 AHB Bridge . . . 38

7.3 Traffic interceptor top 39
7.4 Connector 39
7.5 General info 41
7.6 AHB Slave 41
7.7 SPI Top 42
7.8 MSS Routine 46
7.9 Wireshark Dissectors 48

8 Results **51**
8.1 Hardware Test 55
8.2 Usage results 59
8.3 Timing results 60
8.4 Power results 60

9 Conclusions and Future work **61**

A SmartFusion2 SoC FPGA Architecture **63**

Bibliography **65**

List of Figures

1.1 Integration of Anybus CompactCom with a host.[1] 3

2.1 Methods around the system 6
2.2 Method 1 evaluation 7
2.3 Method 2 evaluation 7
2.4 Method 3 evaluation 8

3.1 Anybus connector.[1] 13
3.2 SPI Mode Anybus connection[2] 15
3.3 3 Wire mode timing diagram.[2] 16
3.4 4 Wire mode transfer example.[1] 17

4.1 SPI Frame.[2] 19

5.1 Anybus CompactCom with traffic interceptor connected. 23
5.2 Anybus CompactCom Expansion Board 023070-B. 24
5.3 Anybus CompactCom M4 ETN. 25
5.4 Expansion Boards where one is cut. 25
5.5 Anybus CompactCom traffic interceptor. 26

6.1 Traffic interceptor detailed diagram. 29
6.2 AMBA AHB 3 masters and 4 slaves. [3] 31
6.3 AMBA AHB Simple transfer [3] 33

7.1 Traffic interceptor block diagram. 37
7.2 Traffic interceptor wrapper. 38
7.3 Traffic interceptor top. 39
7.4 Anybus power up [1] 41
7.5 SPI Top block diagram 42
7.6 Micro controller Subsystem Flowchart 47
7.7 Graphical representation of how a Dissector displays data 49

8.1 Test traffic example used in prototype. 51
8.2 MISO Frame reconstructed from a single frame. 53

8.3 MOSI Message reconstructed from a single frame. 54
8.4 MISO Frame reconstructed from a stream of frames. 55
8.5 Traffic interceptor test hardware 56
8.6 Traffic interceptor test hardware 57

A.1 Microsemi SmartFusion2 FPGA General Architecture.[4] 64

List of Tables

3.1 Pin type of CompactFlash^{TM} connector.[1] 13
3.2 Pin assignation for SPI mode.[1] 14
3.3 SPI interface signals.[5]. 16
3.4 Timing parameters for 3-Wire mode [1] 17
3.5 Timing parameters for 4-Wire mode [1] 18

4.1 SPI Frame MOSI [2] 20
4.2 SPI Frame MISO [2] 20

5.1 Smart Fusion2 M2S025 Overall specifications [4] 27
5.2 M2S025 MicroController Sub System Overall specifications [4] 27

6.1 AHB Bus HRESP commands [3] 32

7.1 Signals intercepted in the connector block 40
7.2 Signals from figure 7.4 [1] 41
7.3 MOSI Frame division in FPGA Logic block. 43
7.4 MISO Frame division in FPGA Logic block. 44
7.5 Anybus CompactCom MOSI Memory Map 45
7.6 Anybus CompactCom Memory Map 45
7.7 Message Header [6] 50

8.1 Test frame parameters. Lengths given in 16 bit words 58
8.2 Hardware test results 59
8.3 Anybus Traffic Interceptor Usage 60
8.4 Timing results 60
8.5 Power results 60

Chapter 1
Introduction

Automated industrial systems often use a distributed topology, where a hierarchy of interconnected systems is used to control a process. Several devices can coexist in this hierarchy, such as Human Machine Interfaces (HMI), Programmable Logic Controllers (PLC), sensors, actuators, motor controllers etc. For such a hierarchy to work, some kind of communication needs to happen between these devices.

A fieldbus is an industrial network system that allows real-time control by connecting different instruments in a plant. Communication requirements, such as bandwidth or determinism, can vary wildly between different processes and even between different parts of the same process (e.g the communication between an HMI and a PLC doesn't have to be time-critical but if a sensor detects an anomaly, a process must stop immediately for safety). Examples of fieldbus networks include Profibus, Modbus and CAN.

With Ethernet being a de facto standard for networking, it seems natural to replace fieldbus networks with Ethernet based systems. However, standard Ethernet has some drawbacks when used in industrial environments, such as lack of determinism and real time control, sensitivity to electrical noise, difficulty to run for long distances, and cost. Nevertheless, its greater bandwidth, and the ability to communicate over routers and servers that work with TCP/IP, as well as the lower cost for the equipment, still make Ethernet desirable for some applications.

In order to bridge the gap between fieldbus networks and Ethernet, different Industrial Ethernet protocols were created. These allow Ethernet to be used in industrial environments by adding determinism and real time control capabilities. Some examples of Industrial Ethernet protocols include EtherNet/IP™, PROFINET and EtherCAT.

The different requirements and lack of a global standard have resulted in multiple non compatible networks being used in the industry, which makes it impossible for a manufacturer to support all existing protocols and ties customers to hardware compatible with the standard that they are already using, often resulting in impossibility to buy equipment from different manufacturers in order to save costs.

To solve these issues, HMS Industrial Networks over the years has proposed plenty of solutions to interconnect different industrial equipment or machinery in multiple ways, such as functional safety, remote access, remote management, and multi-network connectivity among others, however, in this thesis we will be focusing in Anybus, a solution that allows the host equipment to have instant connectivity to all major industrial networks on the market.

Anybus is used nowadays mostly in the factory automation market, but as communication is becoming increasingly important in other areas such as buildings, vehicles, process plants and infrastructure, Anybus is venturing into new markets. Its products consist of electronic hardware and software which enable industrial devices to communicate. The focus for them is to be used as a gateway which provides a translation of different protocols to an internal one, and then, it provides the host access to different industrial network standards.

One way to do this is by embedding a hardware communicator into the host to enable the industrial network connection, this is called Anybus CompactCom.

Anybus CompactCom has different presentations, such as chip, brick and module, each one of them adapts to different needs in a system such as, flexibility or expertise on industrial networks, to mention some.[1]

However, regardless of its presentation and network module, the integration of the CompactCom with a host is implemented in the same way and can be observed in Figure 1.1. One can notice that the signals are connected from the host CPU to interfaces that are later integrated to the Anybus CPU, from there, the converted data is sent to a communication controller that passes it to a physical interface and lastly it is sent to the network. The flow can go both directions, i.e. from the network to the Anybus CPU as well. It is important to mention that, the data that exits the host CPU is handled by software intended to be compatible with Anybus.

Because of the wide variety of industries that the CompactCom can be used, the software to set it up can be very different from case to case, and for that reason errors might occur during its implementation. As mentioned before, the integration of the Anybus CompactCom with a system is done directly with the host CPU and as shown in Figure 1.1. This means that the only output it has is the network, leaving many areas where a problem might occur. One could say that when debugging the system, the CompactCom acts as a black box, due to the fact that the inner process is unreachable at the moment.

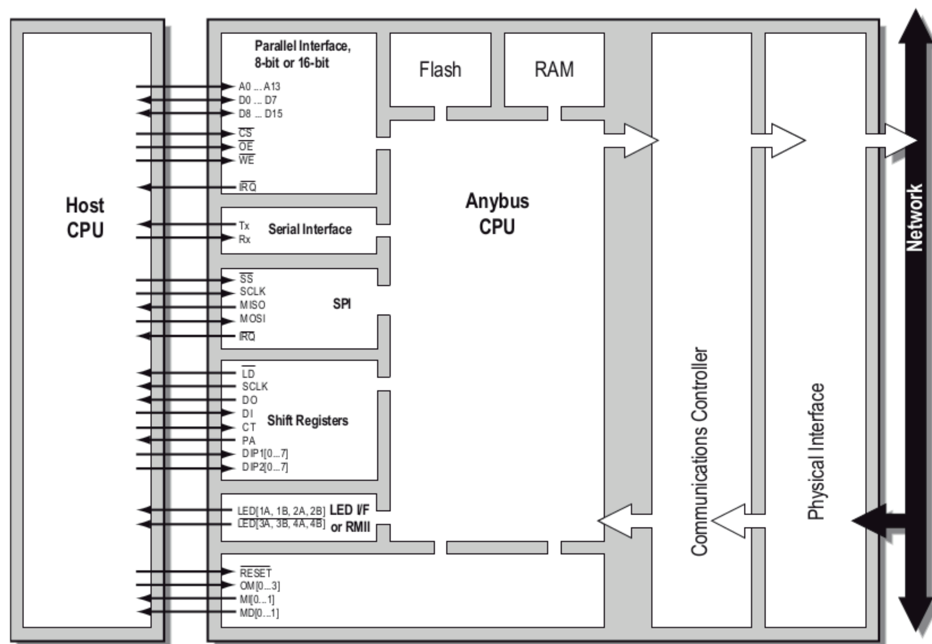

Figure 1.1: Integration of Anybus CompactCom with a host.[1]

Chapter 2

Related Work

Because of the wide variety of industries that the CompactCom can be used, the software to set it up can be very different from case to case, and for that reason errors might occur during its implementation. As mentioned before, the integration of the Anybus CompactCom with a system is done directly with the host CPU and as shown in Figure 1.1. This means that the only output it has is the network, leaving many areas where a problem might occur. One could say that when debugging the system, the CompactCom acts as a black box, due to the fact that the inner process is unreachable at the moment.

The purpose of this thesis is to analize and compare possible solutions that could solve this problem. In order to do that, the comparison to start with will be the actual method and two more, furthermore out of these methods a general to particular analysis will guide to the solution sugessted in this text, a traffic interceptor.

The debugging method at this moment is to send a field engineer to debug the customer's software, that from now on will be considered method 1. What will be considered as method 2 will be a traffic interceptor with an existing known protocol (Ethernet) which will be placed at the output of the CompactCom. Method 3 is proposed to ba a traffic interceptor that can be plugged between the Host and the Anybus CompactCom. Figure 2.1 illustrates the difference between methods in the system, i.e, where is the application of them and in future sections in this chapter why is the place in the system affecting their behaviours.

In order to start this comparison the parameters established were the following:

- Time-consumption: The grading of this parameter will be based in the time needed to generate a final working solution.
- Difficulty: This parameter will be graded according to the complexity that the method demands.
- Effectivity: The grade will be established according to the time that takes to find a solution.
- Reliability: The last parameter will be the accuracy of detecting broken/incorrect packages in the traffic.

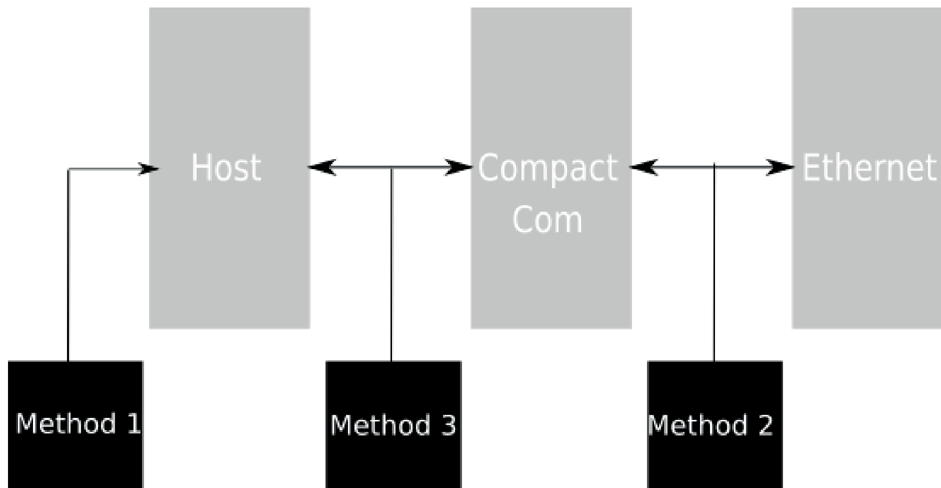

Figure 2.1: Methods around the system

The scale will be set from 1 to 5 and the criteria will be based on the experience of field engineers who have actually worked with host issues before and have experience on them and with the help of the company supervisors feedback.

2.1 Methods comparison

2.1.1 Method 1

As seen in Figure 2.1, method 1 is implemented in the beginning of the system, the way it is working now is very subjective from problem to problem and from one engineer to another. However there is no extra funcionality implemented, for that reason this method is the one with the highest grade in terms of difficulty and in time consumption. When it comes to Effectivity, the grade is the lowest since the time that is not spent in a solution to debug issues in this method, each issue represents a different challenge where the field engineer has to read the lines in the code implemented by the customer, for this reason it is considered very uneffective and has the lowest grade. Finally when it comes to reliability it might work for the tests realized by the field engineer during the the debugging, since they do not cover all the tests, new issues might appear in the future and for this reason the reliability grade is medium. Figure 2.2 shows the plot for the parameters mentioned above.

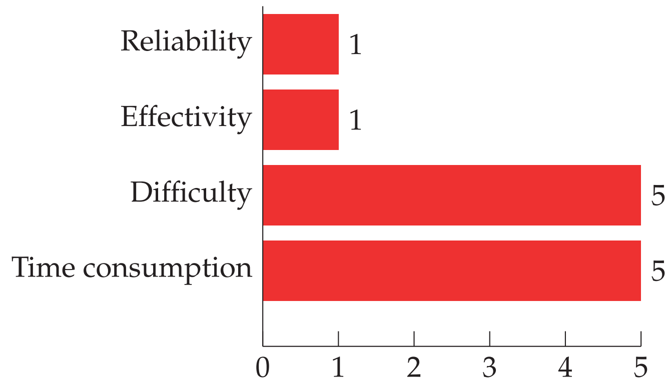

Figure 2.2: Method 1 evaluation

2.1.2 Method 2

When it comes to Method 2, the advantage in other scenarios would be the fact that there are already traffic interceptors that have the Ethernet protocol implemented. However, when it comes to this specific product, this situation is not quite helpful. The reason is that the Ethernet protocol in this system does not provide useful information for debugging since the ethernet protocol is just a way of sending information translated from anybus to Ethernet. Moreover, the Ethernet solution is just one of the protocols the Anybus can translate to. Figure 2.3 shows the grades given to this method.

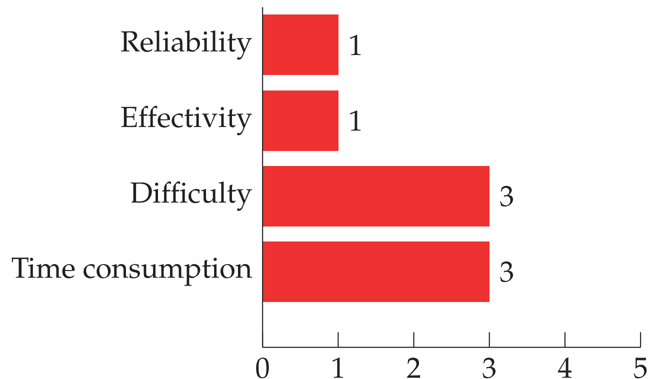

Figure 2.3: Method 2 evaluation

2.1.3 Method 3

Finally, Method 3 is a solution suggested that could shine where the previous 2 would fail. Its first drawback would be the time spent into implementing this, since it requires to modify the existing board to take the internal wires that so far are closed between the host and the Anybus. For this reason this solution has the

lowest grade when it comes to difficulty and time consumption. However, when it comes to realiability and effectivity it is the strongest it is expected to work in regular and in corner case situations. As well in this method, the reliability is the highest since the Anybus protocol checking can be automatized.

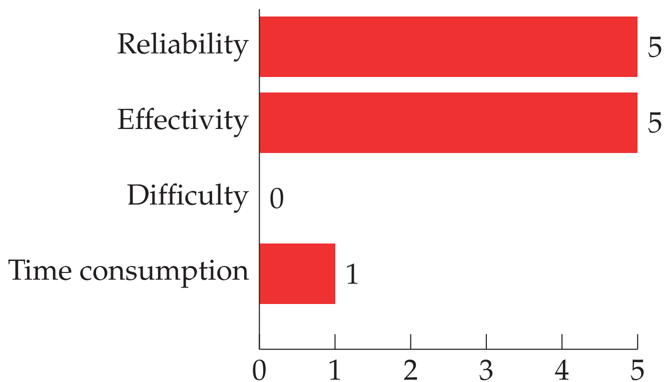

Figure 2.4: Method 3 evaluation

In conclusion, after comparing the three methods with their respective grades, Method 3 offers the best solution since it is a one time work that can offer a much more effective and reliable. However,there are many ways of implementing a traffic interceptor which will be studied next.

2.2 Software and hardware implementation comparison

A traffic interceptor is a device that captures the traffic circulating through a network, allowing the interactions between the connected machines to be stored and analyzed for several purposes, such as troubleshooting, testing, or auditing. In order to do so, the interceptor has access to the physical layer of the network, either the communication wires or the air in the case of wireless networks, and can sense their state to decode the information that is being transmitted. In that sense, a traffic interceptor can be thought of as a network device that only receives data through its connection, unlike a typical network node, which can usually send and receive data using dedicated physical and logical channels. Typically, a traffic interceptor does not interact directly with the network, so the rest of the devices can communicate as usual while the sniffer is "listening".

From an implementation point of view, the traffic interceptor can be seen as having only inputs in its network interface, and another separate interface that allows for the captured data to be transmitted or analyzed. There are two main ways of building a network traffic interceptor or sniffer: a software approach and a hardware approach. In a software approach, the data is decoded by a CPU or

microcontroller, tipically in a computer. This kind of solution is common in Ethernet sniffers, because it has several advantages: first of all, standard PC hardware can be used, making it a simple and inexpensive solution. Existing computers are often used for this, meaning that the cost is basically nil. Another benefit of this method is that the software allows for great flexibility, which makes it possible to decode many different protocols and perform advanced analysis by leveraging existing traffic analyzer tools. For example, Wireshark is a free open source traffic analysis tool, which provides the means for analyzing several communication protocols, and allows for new ones to be added via plug-ins. This tool is often considered the industry standard for traffic sniffing.

Because of the previously mentioned reasons, software-based traffic analyzers are the most commonly used varieties, as they can be used for free by using existing hardware and free software. However, pure software solutions are not suitable for all network analysis needs, due to some of their limitations. The first one is that for networks different than Ethernet, specific hardware will always be needed to interface with the computer: this can be either commercially available hardware for other standard networks (which can amount to a significant cost), or custom hardware solutions. Another limitation is that software based systems usually have a low time accuracy, usually in the order of 1us resolution and uncertainty in the order of miliseconds. For some kinds of networks, specially real time networks with cycle times in the order of miliseconds, this kinds of solutions become unsuitable [7].

Hardware based traffic analyzers, specifically FPGA based ones, can help solve those issues. By using an FPGA, custom digital hardware can be designed without needing to incur the costs of producing an ASIC or a highly complex protocol-specific PCB, both in time and money. FPGAs can implement the logic required to intercept and decode the network traffic, as well as the transmission of the captured data to a computer for further analysis. At the same time, these chips can provide a stable and accurate time base, thanks to their highly deterministic nature. By moving the timestamping to the hardware, time can be measured in a more precise and stable manner. As an example, in [7], the authors managed to capture Ethernet traffic with a resolution of around 100ns and a jitter of 0.1 us, compared to 1us resolution and over 300us of jitter for a software solution.

FPGAs have been widely used to implement interceptors for different protocols thanks to their ability to intercept faster protocols with less dropped packets, which makes them more accurate overall. Some examples of this include a PCI bus sniffer developed by [8], where the authors hifhlight the lower cost and higher flexibility of the FPGA solution. Another example is the network analyzer based on the NetFPGA 1G platform implemented at [9]. NetFPGA is an open platform developed at Stanford to allow researchers and students get hands-on experience with hardware based networking and learn about processing large volumes of packets on fully-loaded Gigabit Ethernet links [9].

Several other protocols have been analyzed using FPGA based analyzers, such as CAN [10] and I2C [11]. All of this examples suggest that a FPGA solution could be ideal for analyzing the Anybus protocol: on one hand, the high

accuracy and speed support would allow it to capture the SPI mode at full speed. On the other hand, the flexibility of the programmable logic means an easier expandability to add the other operating modes in the future.

Chapter 3

Understanding Anybus CompactCom

In order to understand the traffic interceptor, basic knowledge of the Anybus CompactCom system is required, since the traffic interceptor is developed as an aid for the development of compatible products.

In this chapter, the different operating modes and interfaces of Anybus CompactCom will be explored. Although Anybus CompactCom has different versions, this chapter will focus on the M40. The intention of this is to later be able to present different concepts that are used for the realization of the projec.

As it can be observed in Figure 1.1, Anybus CompactCom has 5 different host communication interfaces, corresponding to different operating modes. This figure illustrates the basic properties of these interfaces, signals and their relation to the host application. It is important to notice that only one communication interface is available at a time, which is set up at startup. These communication interfaces will be referred from now on as operation modes, and they will be briefly described as mentioned in [2].

3.1 Anybus CompactCom operation modes

3.1.1 Parallel Interface, 8-bit or 16-bit

This mode is a common 8-bit or 16-bit parallel slave port interface. It can be incorporated into any microprocessor-based system that has an external address or data bus. Its implementation is similar to implementing an 8-bit or 16-bit wide SRAM, including an internal interrupt request line. LED interface signals are not available in this mode.

3.1.2 SPI

SPI stands for Serial Peripheral Interface, and is a synchronous serial link. It operates in full duplex mode and devices communicate in master/slave mode,

where the Anybus CompactCom is always the slave. In terms of its performance, it is slower than the parallel interface, but faster than the serial interface (UART).

3.1.3 Stand-Alone Shift Register Interface

Anybus CompactCom M40 can also work without a host processor, the process data in this operation mode is communicated to the shift registers on the host.

3.1.4 Serial Interface UART

As mentioned before, Anybus CompactCom has different versions, which implement different features or new operating modes. An example of this is the serial interface operation mode, which is included in the M40 in order to provide backward compatibility with the Anybus CompactCom 30. This interface is event based, and it is not recommended to use with an M40 module since it does not take advantage of the performance of this version. However, it is mentioned since it can be used for backwards compatibility.

3.1.5 LED Interface

In almost all the operating modes (excluding 16-bit parallel) it is possible to know the network status by using LED output signals, which can be driven by software.

3.1.6 Reduced Media-Independent Interface (RMII)

This interface is used for Transparent Ethernet. In this case, Industrial Ethernet communication is handled by the Anybus CompactCom and other Ethernet communication is routed to the host applications. LED interface signals are not available in this mode.

Despite the fact that Anybus CompactCom can operate in all the modes mentioned above, the traffic interceptor developed in this thesis only covers the SPI mode, since it is the most common one. The interceptor has been designed with expandability in mind, such that other operation modes can easily be added in the future.

3.2 Physical Connectors

The M40 model uses a 50-pin CompactFlash™ style connector, which can be seen in Fig 3.1. These pins can be of different types, which are defined in Table 3.1.

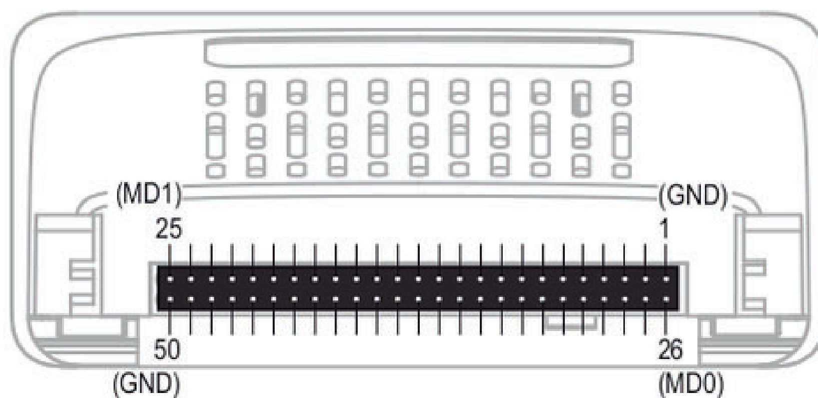

Figure 3.1: Anybus connector.[1]

It is important to acknowledge the pin type since it plays a vital role when implementing the communication between CPUs from the host and the Anybus CompactCom.

Pin type	Definition
I	Input
O	Output
I/O	Input/Output (bidirectional)
OD	Open Drain
Power	Pin connected directly to the power supply, GND or 3.3 Volts

Table 3.1: Pin type of CompactFlash^{TM} connector.[1]

3.3 SPI Pins

The relevant pins vary in different operating modes, but as mentioned before, this project focuses only on the SPI operating mode. For that reason, Table 3.2 shows only the name in which pins will be referred in such mode. Moreover, a description is added in order to clarify the used abbreviations.

Pin	Signal Name	Type	Description
49 24 48 23 47 22 46 21	DIP1_0 DIP1_1 DIP1_2 DIP1_3 DIP1_4 DIP1_5 DIP1_6 DIP1_7	I I I I I I I I	DIP switch. Usage defined by application. Connect to GND if not used
45	$\overline{SS}$	I	Slave select. Active low
20	SCLK	I	Serial Clock Input
44	MISO	O	Master input, slave output. Input to the master's shift register, and output from the slave's shift register.
19	MOSI	I	Master output, slave input. Output from the master's shift register, and input from the slave's shift register.
43 18	(not used)	I O, I	Connected to 3.3 V.
14 39 15 40 16 41 17 42	DIP2_0 DIP2_1 DIP2_2 DIP2_3 DIP2_4 DIP2_5 DIP2_6 DIP2_7	I I I I I I I I	DIP switch. Usage defined by application. Connect to GND if not used
4 29 5 30 6 31 7 32	LED1B LED1A LED2B LED2A $\overline{LED3B}$ $\overline{LED3A}$ $\overline{LED4B}$ $\overline{LED4A}$	O O O O OD OD O O	LED interface. Give access to LED indications.
34 33 10	(not used)	I	Connected to 3.3 V.
9	$\overline{IRQ}$	O	Active low Interrupt Request signal. Asserted by the Anybus CompactCom module.
36 11 35	OM0 OM1 OM2	I	Operating mode for SPI operating mode.
3	OM3/ASI TX	O, Strap	Black channel output. During startup the pin is used to define the operating mode of the module.
28	ASI RX	I	Black channel input. Connect to 3.3 V if not used.
27 2 26 25	MIO/SYNC MI1 MD0 MD1	O O 	
8	$\overline{RESET}$	I	

Table 3.2: Pin assignation for SPI mode.[1]

3.4 SPI Interface Signals

In the previous section, the usage and type of pins involved in the SPI operating mode were shown. In this section, the interface signals and the way this mode

behaves will be explained.

Figure 3.2 shows the connection between Host and Anybus in this Operation mode.

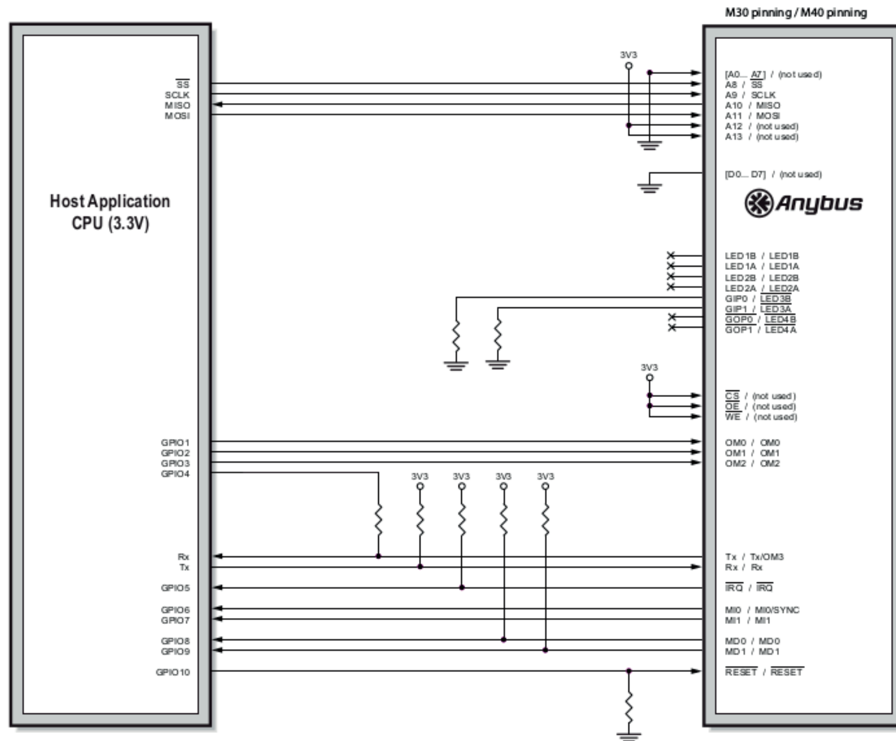

Figure 3.2: SPI Mode Anybus connection[2]

During an SPI transmission, the data is transmitted and received simultaneously. The serial clock synchronizes shifting and sampling of the information on two serial data lines. In the SPI Protocol, transmissions can be made in different combinations of the SPI clock depending on its phase and polarity. Anybus CompactCom works with the clock phase (CPHA) = 0. Clock polarity (CPOL) is also 0. This is known as SPI Mode 0.

A usual transfer with CPHA = 0 is described by Motorola [5] in the following way:

The first edge of the clock line is used to clock the first data bit of slave into the master and the first data bit of master into the slave. The clock output from the master remains in the inactive state for half a period before the first edge appears. Half a cycle later the second edge appears on the clock line. When this second edge occurs the value previously latched from the serial data input is shifted in LSB order. After the second edge, the next bit of the SPI master is transmitted out of the serial data output of the master to the serial input on the slave. This

process continues for a total of 16 edges.

The SPI operating mode uses 3 or 4 signals, depending on the whether the optional $\overline{SS}$ signal is used or not. The signals are presented in Table 3.3, descriptions were obtained from the SPI Block User Guide by Motorola.[5]. When the $\overline{SS}$ signal is in use, the SPI interface is said to be in 4-Wire mode, and in 3-Wire mode when this signal is not used.

Signal	Description
SCLK	Signal used to output the clock with respect to which the SPI transfers data or receive clock in case of slave.
MOSI	Signal used to transmit data out of the SPI module when it is configured as a Master and receive data when it is configured as slave.
MISO	Signal used to transmit data out of the SPI module when it is configured as a Slave and receive data when it is configured as Master.
$\overline{SS}$	(Optional). Signal used to output the select signal from the SPI module to another peripheral with which a data transfer is to take place.

Table 3.3: SPI interface signals.[5].

3.4.1 3-Wire Mode

This mode uses the SCLK, MOSI and MISO signals. The $\overline{SS}$ signal must be tied low permanently. The SCLK signal must be idle high. The modules detect start and stop of a transfer by monitoring the SCLK activity. The downside of this mode is that it is not possible to detect multiple SPI slaves.

Additionally, there must be an idle period of at least 10 microseconds between 2 transfers and the SCLK signal must never remain high for more than 5 microseconds during a transfer. In Figure 3.3 one can observe an example of a transaction realized by using this mode, as well, Table 3.4 shows the minimum and maximum values for the timing during this mode.

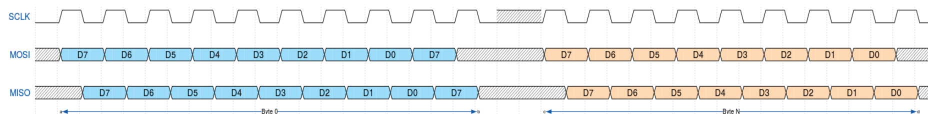

Figure 3.3: 3 Wire mode timing diagram.[2]

Constraint	Min Value	Max Value
MOSI setup before SCLK rising edge	10ns	-
MOSI hold after SCLK rising edge	10ns	-
MISO change after SCLK falling edge	0ns	20 ns
SCLK low period	20ns	-
SCLK high period	20ns	-
SCLK period	50ns	-

Table 3.4: Timing parameters for 3-Wire mode [1]

3.4.2 4-Wire Mode

In 4-wire mode the transfer of data works in a similar way as in 3-wire mode: MOSI data is sent on SCLK rising edge and MISO is sent on the SCLK falling edge. However, here a new signal, $\overline{SS}$ (Slave Select), is implemented. It acts as an indicator for the start and stop of a transfer. In this mode the SCLK signal can be either idle high or idle low. This mode also allows multiple SPI slaves on the same bus. It is worth mentioning that the MISO signal is tristated when $\overline{SS}$ is high, in order to avoid collision with other slaves.

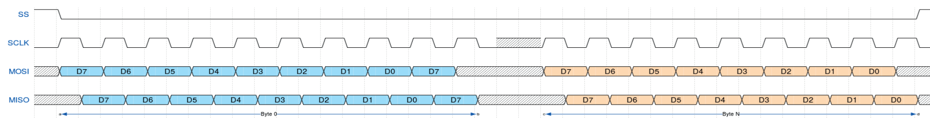

Figure 3.4: 4 Wire mode transfer example.[1]

In Figure 3.4 data transfer in SPI 4 wire mode can be observed. As long as the $\overline{SS}$ signal remains low, the data sent will belong to the same transfer as seen in this Figure. In addition to this, Table 3.5 shows the time constraints that this mode demands.

Constraint	Min Value	Max Value
MOSI setup before SCLK rising edge	10ns	-
MOSI hold after SCLK rising edge	10ns	-
MISO change after SCLK falling edge	0ns	20 ns
SCLK low period	20ns	-
SCLK high period	20ns	-
SCLK period	50ns	-
$\overline{SS}$ setup before first SCLK rising edge	20ns	-
$\overline{SS}$ hold after last SCLK rising edge	20ns	-
MISO valid after falling edge of $\overline{SS}$	-	20 ns
Miso high-z after rising edge of $\overline{SS}$	-	20 ns

Table 3.5: Timing parameters for 4-Wire mode [1]

Overall, this is a brief description of how Anybus CompactCom works in SPI Mode, which is the most commonly used operating mode. However, in order to describe the implementation of the Traffic interceptor in this thesis, more concepts need to be introduced. Future chapters will discuss how these operating mode is used to trasmit data, and how this data can be processed and analyzed. A tool that allows the user to interpret the data that will be produced by the traffic interceptor will be introduced as well.

Chapter 4

SPI Frame

In the previous chapter, the SPI protocol was introduced and briefly described, as well as the way it is implemented inside of the Anybus CompactCom. However, the CompactCom module shapes the data received in any operation mode as Anybus frames and, for that reason, the SPI operation mode frame will be described in this chapter. This will help the reader understand how the traffic interceptor stories data, and the architecture of the traffic interceptor, which will be discussed in later chapters.

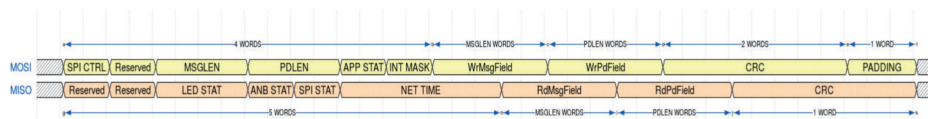

Figure 4.1: SPI Frame.[2]

During SPI operation, most bytes are transmitted with the most significant bit first, but the byte order is little endian, [2] meaning that the least significant byte is transmitted first. The only exception is the CRC32 checksum field, which is transmitted in big endian order. Figure 4.1 shows how frames are sent. Here, it can be noticed that both MOSI (Master Output Slave Input) and MISO (Master Input Slave Output) lines have a frame of their own, similar in structure, but not identical. Table 4.1 presents the MOSI Frame components, and Table 4.2 introduces the ones that belong to the MISO.

Byte	Name	Description
0	SPI Control	Described in Section 4.1.1.
1	Reserved	Confidential to HMS.
2-3	MSGLEN	Indicates the size of the message field in words.
4-5	PDLEN	Indicates the size of the write process data field in words.
6	APP STATUS	Described in Section 4.1.2.
7	INT MASK	Described in Section 4.1.3.
MSGLEN words	MSGFIELD	Message field.
PDLEN words	WRPDFIELD	Write process data field.
2 word	CRC	Error checker.
1 word	PADDING	Dummy data. Ignored by anybus

Table 4.1: SPI Frame MOSI [2]

Byte	Name	Description
0	Reserved	Confidential to HMS Industrial Networks.
1	Reserved	Confidential to HMS Industrial Networks.
2-3	LED STATUS	Described in Section 4.2.1.
4-5	ANB STATUS	Described in Section 4.2.2.
6	SPI STATUS	Described in Section 4.2.3.
7	NET TIME	Lower 32-bits of the network time.
MSGLEN words	MSGFIELD	Message field.
PDLEN words	RDPDFIELD	Read process data field.
2 word	CRC	Error checker.

Table 4.2: SPI Frame MISO [2]

4.1 SPI MOSI Frame

In this section, the relevant components of the SPI MOSI frame will be described briefly. Some of them will be further expanded when the traffic interceptor is introduced.

4.1.1 SPI Control

This section of the MOSI frame contains vital information to control the contents inside of the frame, such as message and data. A more detailed explanation of the bits in this byte will be given later in this document.

4.1.2 Application Status (APP STATUS)

This byte is used for to indicate the status of the host application. The application sets the code and the Anybus accepts it and handles the situation. Most of the codes denote errors between the host and the Anybus.

4.1.3 Interrupt Mask (INT MASK)

This byte allows the application to enable or disable individual interrupts inside the Anybus. Each bit inside of it acts as a mask to different interrupts inside the Anybus.

4.2 SPI MISO Frame

4.2.1 LED Status

Used as a register, it reflects the current LED status, i.e. represents the value of the LED in the Anybus object.

4.2.2 Anybus Status (ANB STATUS)

This register shows the current Anybus state and supervised bit indicated by the Anybus model. This is a 16-bit register, however, only the lower 4 bits are implemented, and the rest are hardwired to zero.

4.2.3 SPI Status

Similar to SPI Control, this byte is used to denote SPI signals status. It is used as a register and will be described more in detail in further chapters.

After an introduction was given to the SPI frames that the traffic interceptor will work with, this concepts will be described more in detail in further chapters.

Chapter 5

Traffic Interceptor Overview

After describing the Anybus CompactCom and the Anybus protocol, the traffic interceptor can be introduced. In this chapter, the general description of the traffic interceptor and the modules needed for it to be implemented will be discussed.

5.1 General Overview

The Anybus traffic interceptor is a debugging tool that intercepts the traffic from the host's output and input, as shown in Figure 5.1. This device is connected in parallel to the bus that communicates the host and the CompactCom module, allowing the user to see traffic in real time.

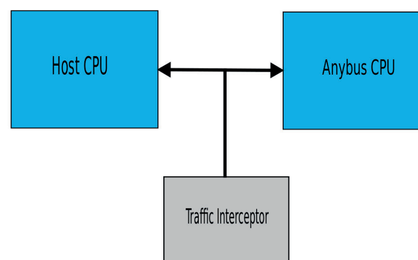

Figure 5.1: Anybus CompactCom with traffic interceptor connected.

Note that the traffic interceptor's purpose, as it's name implies, is only to observe the traffic itself, and it is not intended to fix errors (if there is any) in the process. The goal is to point traffic errors and show the raw traffic, so the user can debug the communication between the CompactCom and the host application.

5.2 PCB Description

Figure 5.1 shows the ideal way to connect the traffic interceptor so it can capture the traffic between both CPUs. In order for this to happen, a special connector is needed.

This connector was provided by HMS Industrial Networks, and it will be briefly described.

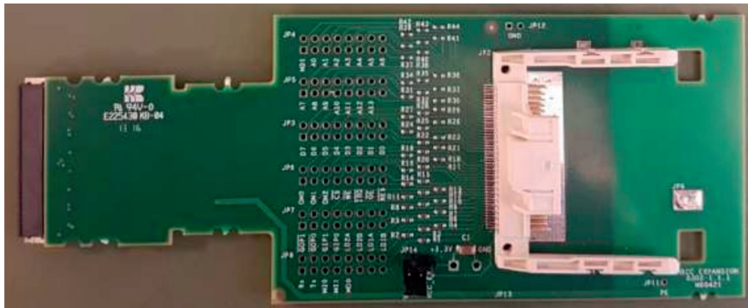

Figure 5.2: Anybus CompactCom Expansion Board 023070-B.

In order to build an Anybus CompactCom Traffic Interceptor, 2 Anybus CompactCom 023070-B Expansion Boards were used (see Figure 5.2). One of the boards was used as a connection expander between the CompactCom module and the host, from which all the bus signals could be tapped. In the other Expansion Board, the left part was removed as seen in Figure 5.4, and the right socket was used to connect the Anybus CompactCom M4 ETN (Figure 5.3), where the traffic interceptor was implemented.

Before connecting them, some modifications need to be done. These modifications are:
On the cut-off board:

- R1, R14 and R44 were removed.
- From R2 to R13 and R15 to R43 the values were replaced to 120R 0603 1/10 W.
- Jumper moved to `VCC_EXT`.

On the whole board:

- Remove the external power conector JP13.
- Jumper moved to `CVV_INT`.

On the ABCC M40 board:

- One entry in the list

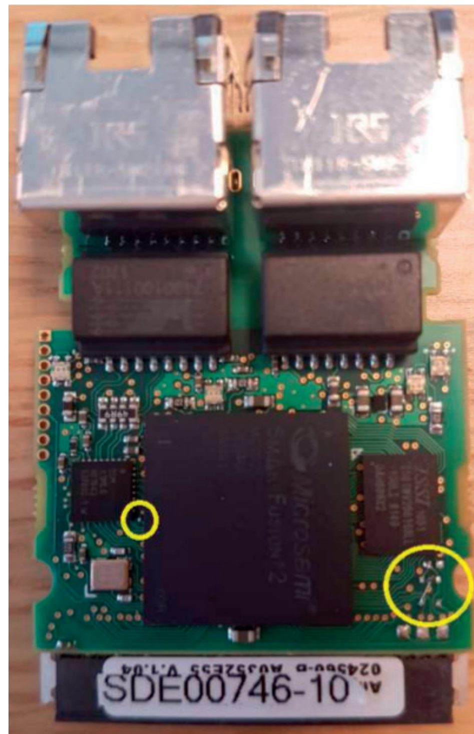

Figure 5.3: Anybus CompactCom M4 ETN.

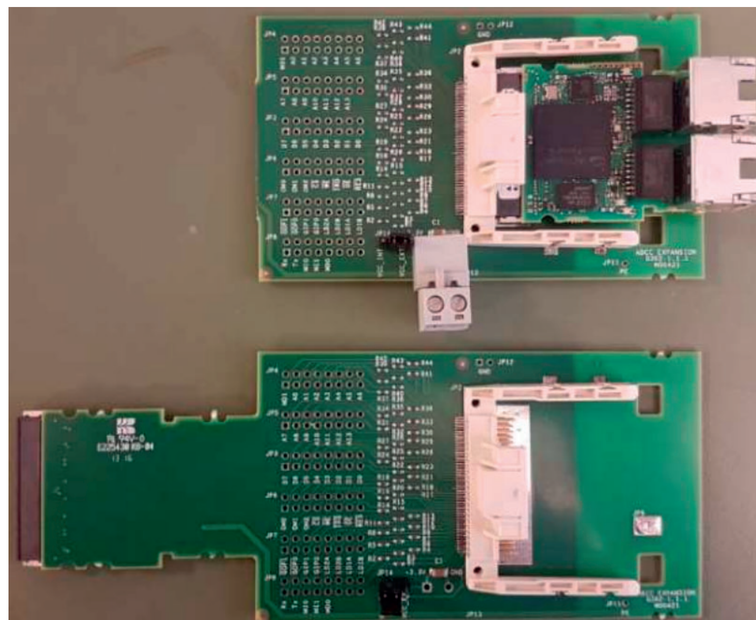

Figure 5.4: Expansion Boards where one is cut.

- Another entry in the list

6 DW-08-14-T-D-1000 Samtec connectors are also needed to join the boards.

Moreover, a standing 120Ω resistor was added to R14 on the cut-off board. The top of the resistor was connected with a wire to the ABCC M40 ETN module TP2. Enabling the traffic interceptor to operate on the DUT reset signal.

Also, `SW_PUSHBUTTON` was added on the cut-off board with one side connected to pin 2 (GND) and to R14.

Lastly, a standing 2200Ω resistor was soldered to R14 and to VCC (3.3V). The result, can be observed in Figure 5.5, by using this hardware the Traffic interceptor can be added into the regular traffic flow.

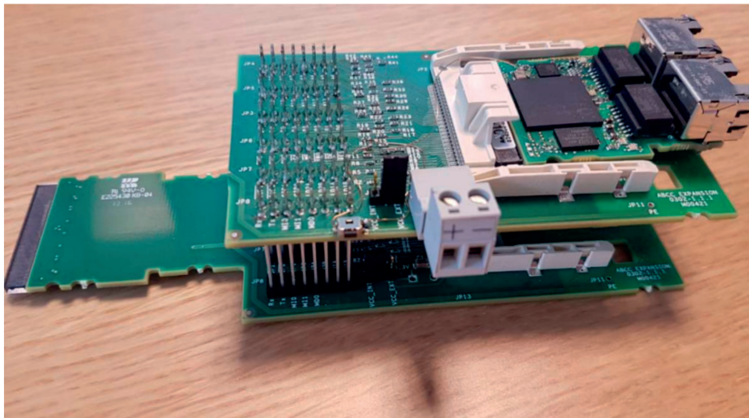

Figure 5.5: Anybus CompactCom traffic interceptor.

5.3 FPGA

In section 5.2 the device where the traffic interceptor will be implemented was described. This section will describe the FPGA contained in the ABCC M40 ETN module, since this will be relevant for the implementation of the traffic interceptor, as well as the results in terms of area, speed and power.

The FPGA that was used in this project was the Smartfusion 2 M2S025. It is manufactured by MicroSemi and it is ideal for general purpose functions. The general architecture of the family can be observed in Appendix A, where it can be seen that if includes a MicroController Sub System, which will be referred to from now on as MSS. Some functionality was implemented in this microcontroller, but this section will focus on the FPGA, where the main logic of the traffic interceptor will be implemented.

The Smartfusion 2 also includes on-chip RAM, which will be important for the implementation of the project, as it will be where the information will be stored. Note that the MSS communicates with the FPGA is by using an AHB bus,

which is a communication protocol that will be described in section 6.2.

Feature	Description
Maximum logic elements	27 696
Math-blocks (18x18)	34
Fabric interface controllers	1
PLLs and CCCs	6
LSRAM 18 K blocks	31
uSRAM 1K blocks	34
Total RAM (Kb)	592

Table 5.1: Smart Fusion2 M2S025 Overall specifications [4]

Table 5.1 shows a few technical specifications that are of interest to the project, regarding area and memory storage. Table 5.2 focuses in the important MSS features that were used for this thesis. This data will be used once again in Chapter 8 to discuss the results in terms of resource utilization apart from the functional results.//

Feature	Description
eNVM(KB)	256
eSRAM (KB)	64
10/100/1000 Ethernet	1
Timer	2

Table 5.2: M2S025 MicroController Sub System Overall specifications [4]

After describing the physical part of the system, the following chapters will discuss different design possibilities and the implementation of the chosen solution.

Chapter 6

Traffic Interceptor Design

One of the most common questions when designing embedded systems, and one of the most complex as well, is related to the partitioning of different tasks across the system. How much of a "Hardware" or "Software" solution will the system be? In reality there is no clear answer for that, since the definition of embedded systems is so wide that the answer in most of the cases is "it depends".

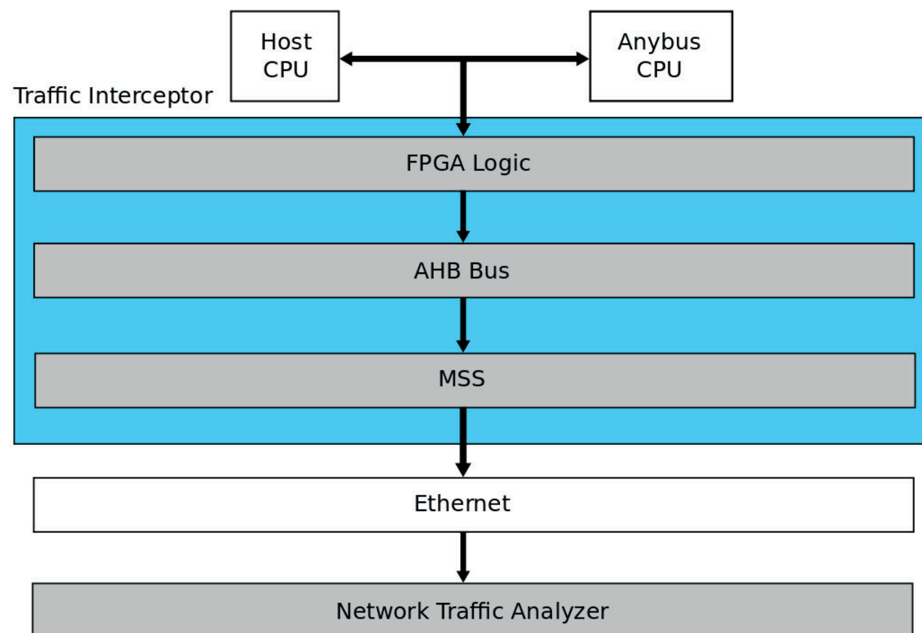

Figure 6.1: Traffic interceptor detailed diagram.

In this case, the complete system used to implement the traffic interceptor can be observed in Figure 6.1. Each part of the system will be described in the following sections.

6.1 FPGA Logic

As mentioned in Chapter 7.1 a big part of the system is the logic that implemented in the FPGA fabric. This allows low level handling of data by using logic gate arrays. In this part, there are RAM memory blocks available as well.

6.2 AHB Bus

The system bus to communicate with other devices, will be described in this section, based upon its specification [3].

The AHB protocol belongs to the AMBA family of communication protocols, which are intended to address the requirements of high-performance synthesizable designs. It has the following features:

- Burst transfers.
- Split transactions.
- Single cycle bus master handover.
- Single clock edge operation.
- Wider data bus configurations (64/128 bits).

A typical AHB bus is designed to be used with a central multiplexer interconnection scheme. Using this scheme, all bus masters drive out the address and control signals indicating the transfer they wish to perform and the arbiter determines which master has its address and control signals routed to all the slaves. A central decoder is also required to control the read data and response signal multiplexer, which selects the appropriate signals from the slave that is involved in the transfer. Figure 6.2 illustrates the structure needed to implement an AMBA AHB design with 3 masters and 4 slaves.

An overview of the AHB operation would be as following:
In order for the AHB transfer to start, the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master will be granted use of the bus.
A granted bus master starts an AHB transfer by driving the address and control signals. These signals provide information on the address, direction and width of the transfer, as well as an indication if the transfer forms part of a burst. Two different forms of burst transfers are allowed:

- Incrementing bursts, which do not wrap at address boundaries.
- Wrapping bursts, which wrap at particular address boundaries.

A write data bus is used to move data from the master to a slave, while a read data bus is used to move data from a slave to the master.

Every transfer consists of:

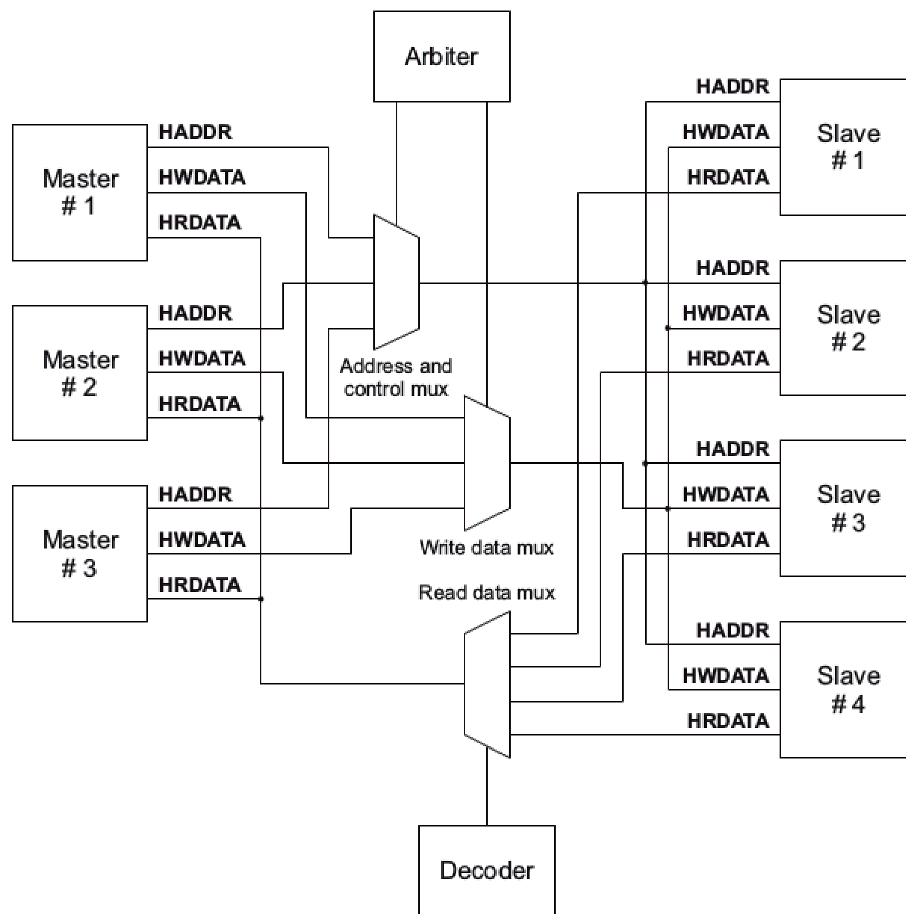

Figure 6.2: AMBA AHB 3 masters and 4 slaves. [3]

- An address and control cycle.

- One or more cycles for the data.

The address phase cannot be extended and therefore, all slaves must sample the address during this time. The data phase, however, can be extended using the HREADY signal. When LOW, this signal causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data. During a transfer the response signals HRESP[1:0] can have the status shown in Table 6.1.

OKAY	Used to indicate that the transfer is progressing normally and when HREADY goes HIGH the transfer has completed successfully.
ERROR	Indicates that a transfer error has occurred.
RETRY and SPLIT	They indicate that the transfer cannot complete immediately, but the master should continue to attempt the transfer.

Table 6.1: AHB Bus HRESP commands [3]

In normal operation, a master is allowed to complete all the transfers before the arbiter grants another master access to the bus. However, it is possible for the arbiter to break up a burst, and in such cases the master must re-arbitrate for the bus in order to complete the remaining transfers in the burst. Figure 6.3 shows a basic transfer that consists of two distinct sections:

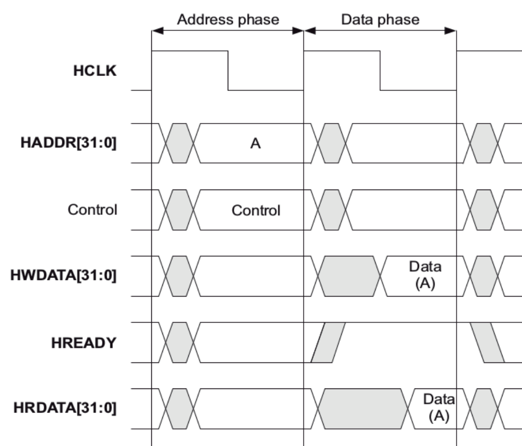

Figure 6.3: AMBA AHB Simple transfer [3]

- The address phase, which lasts only a single cycle.
- The data phase, which may require several cycles. Achieved using the HREADY signal.

6.3 Microcontroller Sub System (MSS)

The micro controller subsystem (MSS) contains a high-performance integrated Cortex-M3 processor, running at up to 166 MHz. The MSS contains an 8 K byte instruction cache to provide low latency access to internal eNVM and external SRAM memory. The MSS provides multiple interface options to the FPGA fabric in order to facilitate tight integration between the MSS and user logic in the fabric.[4]

In this part of the system is where the signals can be converted from pure logic signals into higher level constructs, as well as, routing the data to the Ethernet output to continue its flow.

6.4 Ethernet

As seen in Figure 5.3 the data that is sorted by the traffic interceptor is packaged and sent through Ethernet. This was suggested by HMS Industrial Networks in order to be able to see the traffic with assistance of a computer and using a network traffic analyzer software such as Wireshark.

6.5 Network Traffic Analyzer

The last component of the traffic interceptor, which is not really part of the system (physically) but it is where the data can be viewed and analyzed. As mentioned before, the specific software selected for this task was Wireshark.

After describing each specific component that integrates the traffic interceptor, it is interesting to point out that due to the flexibility that this components offer, it is possible to find several solutions to the same problem. However, the pros and cons were evaluated in order to develop the most complete one. During the first stages of this project 3 different of the proposed solutions were considered. These solutions will be presented using the internal name that they were given, as well as, their respective advantages and disadvantages.

6.6 Software Oriented design

The first solution that was proposed was one that was named "Software oriented design". In this, as its name implies, the focus would be on the software development using the MicroController Sub System. The idea of this is that from all the parts that make up the system, the one that allows an easier maintenance is the MSS. In this solution, the purpose of the logic FPGA logic was to pass the signals in a serial way, then, the data would be reconstructed and reorganized in the microcontroller. Once the data was ready, it would be sent to the Ethernet and then displayed by the network packet analyzer software so the user could make use of it.

The main advantage of this solution is that, as mentioned before, the bulk of its development would be in C code, allowing to simplify its maintenance and readability (in comparison with the FPGA logic block where a hardware description language is used). However, the main downside of this was also the reason why it was not the selected option. It is known that purpose-built implementations in digital logic are usually faster than software ones, and when it comes to intercepting real time traffic, a software solution can be too slow to handle the throughput and, as a result, data can be lost.[7]

In this solution, the intention to simplify the logic in the FPGA becomes the biggest disadvantage for the system, since it would only act as a bridge to the MSS where the data would be sorted, reconstructed and lastly packaged and sent through the Ethernet port. However, as mentioned in previous chapters, it is expected that the SPI operating mode of the Anybus CompactCom sends 1 bit per cycle, with some specific bits being of vital importance to show the status of different blocks in the Anybus CompactCom or the Host implementation. The problem becomes more evident if the only way to communicate the FPGA logic block with the MSS is through memory copy where the FPGA logic block would be updating in the same address bit over bit. In case that the software needs more than one cycle to do any kind of operation with this bit, then data would be lost

and the traffic interceptor would not be able to retrieve all traffic between the devices that are being intercepted. It is worth mention that aim of this project is to get a 0% of packets lost.

6.7 Network Packet Analyzer Oriented design

Another possible design would have been a solution where the main focus would be on the network packet analyzer. This could be implemented thanks to the possibility that Wireshark offers of expanding the protocols to analyze in the way of a plugin. The advantages are similar to the one in Section 6.6, where the development focuses in one part of the system and the rest becomes simpler and acts just as a link for the signals/data to flow through the system.

In this system, the signals would flow directly from the FPGA logic block and written to the memory where the MSS could take them, concatenate them and lastly package them and send it through the Ethernet port. Once the data is in the Ethernet port, Wireshark can capture it and process it.

This in theory could simplify by far the implementation of the traffic interceptor, however this solution is the one that contains the most disadvantages. The biggest disadvantage comes from the way in which plugins (also known as Dissectors) in Wireshark, dissect the byte streams and present them by section. The issue is that in order for the dissectors to work properly, the byte streams need to be of a specific length and correctly organized. In other words, any possible solution needs to send a pre-organized data stream with everything in its corresponding place.

Because of a lack of data sorting this solution is not the best one, and another solutions need to be proposed.

6.8 Logic Oriented design

After seeing the downsides of the previous solution, the authors proposed a third option that is believed to be a better solution. This works by implementing the traffic interception and decoding into the FPGA logic block. The advantages are many and the disadvantages are few and all of them will be discussed.

The first advantage is that the FPGA offers more speed than software as mentioned before, which minimizes the chances of losing a packet. Also, the possibility of organizing the data in the memory before it reaches the microcontroller vastly reduces the amount of processing needed on the software side. This means that the delay introduced by the microcontroller is also minimized, as it just needs to pass through the pre-organized data. Lastly, as mentioned before, the dissectors developed in Wireshark expect the data to come in a specific way in order

to display it correctly. This solution inherently fixes this issue, as the frames get reconstructed in FPGA memory by design.

The main disadvantage of this alternative is its maintenance, since the implementation of the main core in this solution is done in VHDL, which is a language that is usually regarded as less intuitive than C. Apart from that, no other disadvantages where found, so this solution was chosen to be implemented as the Anybus traffic interceptor. The specifics of the implementation, from the FPGA, MSS and Wireshark point of view, will be discussed in the following chapters.

Chapter 7

Traffic Interceptor Implementation

As mentioned in Chapter 6, it was decided to implement the traffic interceptor combining the FPGA fabric and the MSS, given that it would provide the best timing performance and it had virtually no disadvantages. The FPGA logic consists of various blocks that capture and decode the data, which is sent to the microcontroller, where it is formatted and sent via Ethernet, where it can be analyzed with a network analyzer software. This chapter will discuss the detailed implementation of the different parts of the system, and the performance and implementation results will be shown in the following chapters.

Figure 7.1 shows the architecture for the traffic interceptor.

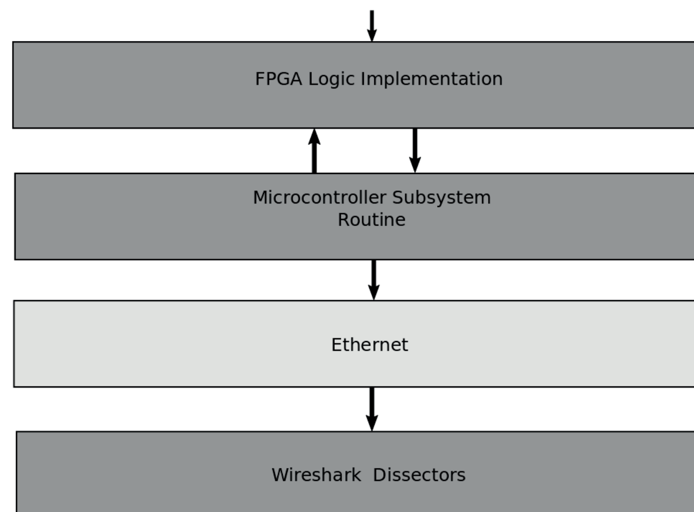

Figure 7.1: Traffic interceptor block diagram.

7.1 Traffic interceptor wrapper with MSS

This block can be seen with further detail in Figure 7.2. It acts as a wrapper for all the blocks in lower levels. Here is where all the signals are interconnected, including the micro controller sub-system. This means that this is the highest level of the design that resides inside of the AnybusCompactCom M40 ETN that was described before in Chapter .

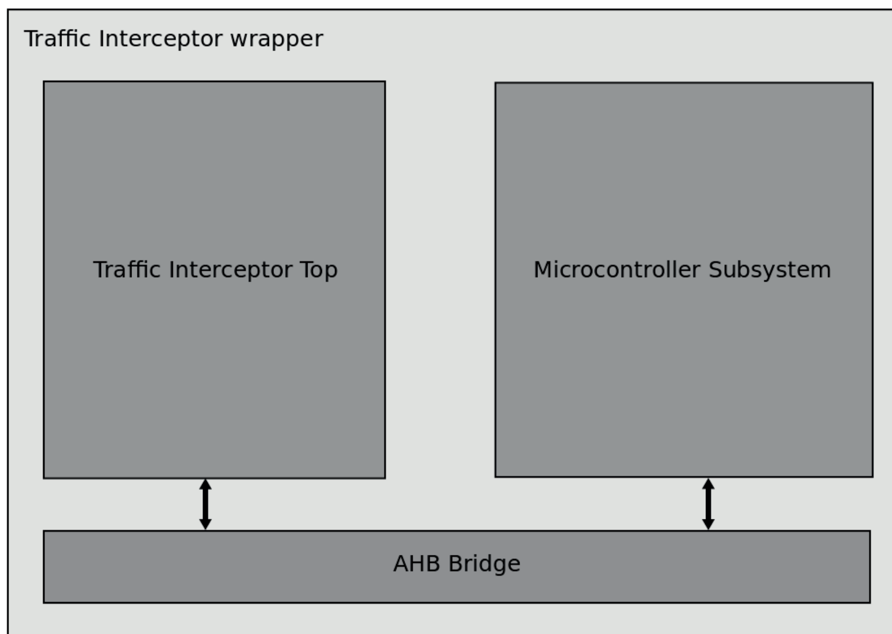

Figure 7.2: Traffic interceptor wrapper.

This wrapper includes the traffic interceptor top module, where all the hardware logic is contained, and it instantiates the MSS and the AHB bridge required for communication between them.

7.2 AHB Bridge

This module allows the connectivity to an AHB master. For it to work properly, it divides the address spaces between the different slaves and, depending on the address given by the AHB Master it generates the signals needed to communicate with the desired AHB slave, that way, both master and slave can send and receive the data from the other part.

7.3 Traffic interceptor top

This block, showin in Figure 7.3 acts as a wrapper for all the sub-blocks that implement the different parts of the traffic interceptor hardware.

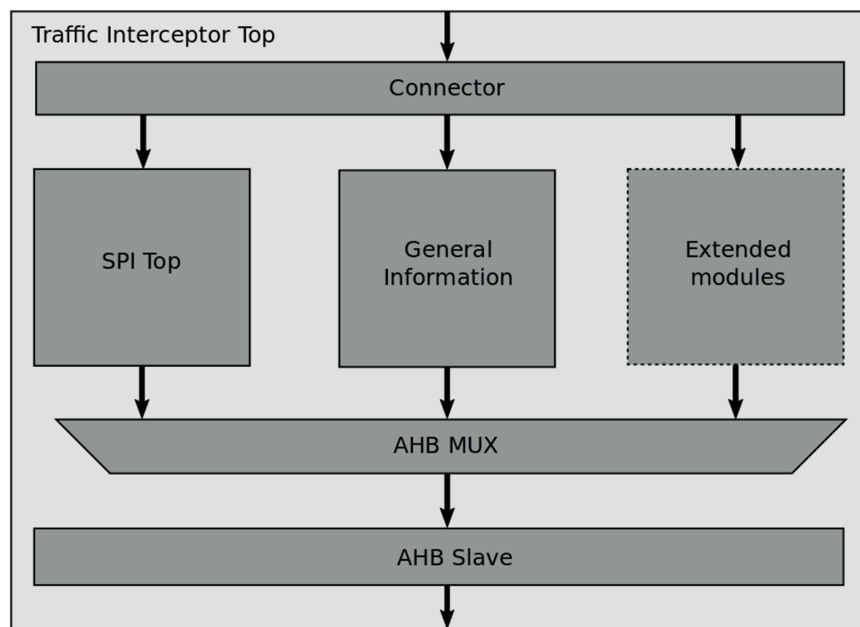

Figure 7.3: Traffic interceptor top.

7.4 Connector

One of the interesting features of a traffic interceptor is that one can capture any signal that the Anybus or Host have either as an input or output. This allows to not only intercept network data flowing in its different operating modes, but also interesting signals that can let the user know the status of different and independent features inside of the system. That is the case for signals that provide additional information, such as the ones appearing in Table 7.1. This kind of information is of great importance for the traffic interceptor, the reason being that even though one operating mode is currently implemented, the system is designed with the future addition of other modes in the future. This signals would allow the interceptor to know which mode is active, as well as other information about the status of the CompactCom module.

Signals	Use
Operating mode	Allows the traffic interceptor to know which module to use for intercepting data. Allows the user to be aware of the operating mode in use.
Interrupt	Displays when an interrupt is asserted.
Sync	Displays the sync status
Led pins	Displays the status of the LEDs in the CompactCom module.
Reset	Displays the status of the reset in the Anybus CompactCom. NOTE: the traffic interceptor has a separate and indipendent reset.

Table 7.1: Signals intercepted in the connector block

This block synchronizes the intercepted signals from the Anybus domain to the FPGA clock domain, using a double register synchronizer for each bit. This is done to reduce the chance of signal metastability issues due to the non-synchronized input signals.

A special case applies for the operating mode signals, which together form an array named "OM" by the HMS Industrial networks documentation [1]. In this array, the last signal has a shared purpose, for that reason, it takes its value before the reset is set high (Reset is active low). After it is set high, Anybus CompactCom gives another use to this pin. Figure 7.4 gives a graphical representation of how the reset process is realized. The meaning of the variable presented in this figure can be observed in Table 7.2.

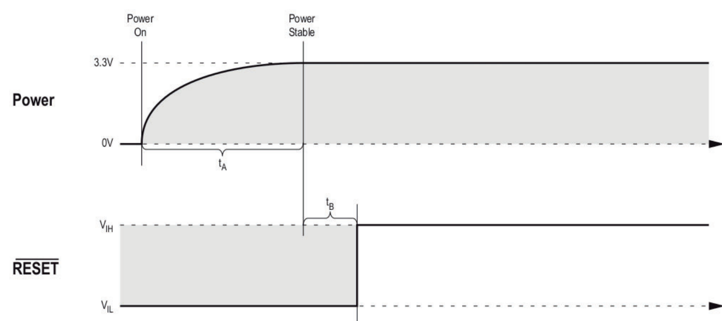

Figure 7.4: Anybus power up [1]

Symbol	Min.	Max.	Definition
t A	-	-	Time until the power supply is stable after power-on; the duration depends on the power supply design of the host application and is thus beyond the scope of this document.
t B	1ms	-	Safety margin.

Table 7.2: Signals from figure 7.4 [1]

7.5 General info

This block is intended to store the extra signals that are not part of the data being transmitted in the bus, which come from the Connector block. They are stored in a register that is updated every time its value changes. The register is then connected to the microcontroller sub system in this design via the AHB slave, to be sent only when it is requested, since this information is treated independently of the rest of the data handled in the operating mode.

7.6 AHB Slave

In order to implement a correct communication with the MSS and, as explained in Section 6.2, the AHB protocol requires the communication to happen between a master and a slave.

In this case, the micro controller sub system acts as a master and is the one that demands data when it is needed in the software routine. This slave is the one

in charge of delivering the data inside the memories in the other modules to the MSS.

The AHB Slave implemented in this module is directly connected to a multiplexer that can be set up to change between modules, as for the moment, this multiplexer selects between the SPI top and the general information register, it was designed that way so it can allow further expansion.

7.7 SPI Top

This block containts the SPI operating mode logic. In this module, all the blocks that cooperate in order to intercept and decode the traffic in the SPI operating mode are instantiated. The SPI top is only active once the corresponding operating mode is detected in the connector block. If a different operation mode is detected, for example when using a hypothetic future expansion, this block does not get activated.

The block diagram for this block can be seen in Figure 7.5.

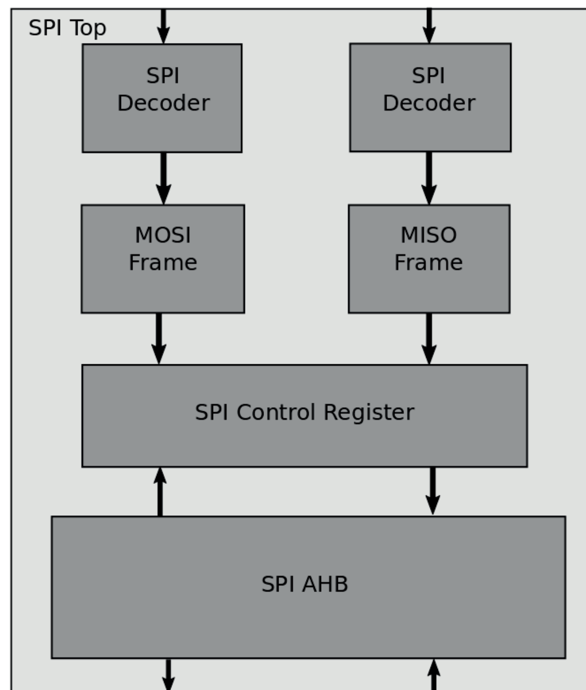

Figure 7.5: SPI Top block diagram

7.7.1 SPI Decoder

The first block that receives intercepted data is the SPI Decoder, which is intended to decode the serial signals from the SPI interface from bit into bytes and words that correspond to the structure each frame requires.

As mentioned in Chapter 4, the MISO and MOSI data is captured in different edges of the SPI clock. The decoder block can be configured to follow the rising or the falling edge of SCLK, which allows it to be reused twice in this system by configuring each instance to the desired mode.

This module works by shifting the input bits on the selected clock edge, and once 1 word (2 bytes) has been saved, it sends it to the next block to be organized into a frame.

7.7.2 MOSI and MISO frames

These blocks are explained together since they are very similar. Their function differs just in the parts of each frame. Table 7.3 shows the fields in the MOSI frame, and Table 7.4 shows the same for the MISO frame.

In the specific case of the MISO frame, it is possible to observe that there are no message length nor process data length fields. The reason is that they are not included in the frame, instead, their values are shared from the MOSI frame, as per the Anybus specification. Because of this, both frames have the same lengths in the two sections.

MOSI Frame
IDLE
SPICTRL & RESERVED
MSGLEN
PDLEN
APPSTAT & INTMASK
WRMSG
WRPD
CRC1
CRC2
WAIT FOR IDLE

Table 7.3: MOSI Frame division in FPGA Logic block.

Another need for these modules is to reorder the words since each byte is transmitted with the most significant bit first, but the byte order inside a word is little endian. For that reason each word is reorganized in the correct order in

whatever address is required for each part.

The Idle and Wait for Idle states are the only ones not corresponding to frame fields, and are included as resting states for when there is no data ready.

MISO Frame Block
IDLE
RESERVED
LED STATUS
ANB & SPI STATUS
NET TIME1
NET TIME2
RDMSG
RDPD
CRC1
CRC2
WAIT FOR IDLE

Table 7.4: MISO Frame division in FPGA Logic block.

7.7.3 SPI Control Register

This block acts as a register for SPI status communication with the MSS. This is where the frame status is stored, and it includes the possibility to be cleared from both sides.

The MSS reads if there is a new frame and it writes that this specific frame has been read from this register. The main intention for this module is to let the microcontroller know when there is an incomplete frame so the user can be aware of it immediately.

7.7.4 SPI AHB

This block is the biggest one in terms of area, the reason being that this is where the data previously handled is stored. This block generates two dual port RAM memories, which are implemented with RAM resources available in the Microsemi SmartFusion2 FPGA.

Data	Address
Anybas CompactCom General Info	0x00000000
MOSI Frame Starting address	0x00000001
MISO Frame Starting Address	0x00000C09
SPI Control & MOSI Reserved	0x00000001
Message Length	0x00000003
Process Data Length	0x00000004
Application Status & Internal Mask	0x00000005
Write Message Field	0x00000006
Write Process Data Field	0x00000005 + Message Length
CRC 1	0x00000005 + Message Length + Process Length
CRC 2	0x00000005 + Message Length + Process Length + 1

Table 7.5: Anybus CompactCom MOSI Memory Map

Data	Address
MISO Reserved	0x00000009
LED Status	0x0000000A
Anybus Status & SPI Status	0x0000000B
Net time	0x0000000C
Read Message Field	0x0000000D
Read Process Data Field	0x0000000D + Message Length
CRC 1	0x0000000D + Message Length + Read Process Data Length
CRC 2	0x0000000D + Message Length + Read Process Data Length + 1

Table 7.6: Anybus CompactCom Memory Map

In Tables 7.5 and 7.6 it is possible to observe the memory maps that were used for this traffic interceptor. Something that may stand out is the fact that there are a couple of dynamic addresses. They are expected to be this way since the message

and process data part of each frame can variate in size and the words are written sequentially in order of arriveal. However it is possible to know the specific address thanks to the message length division in the MOSI frame. To avoid synthesis and place and routing issues, the memory has a maximum size available that is aimed to the maximum length for each of the dynamic parts (message and process data).

The RAMs have 8192 addresses with a word length of 16 bits. The word size was chosen to simplify the writing and reading, since the base size of Anybus fields is 16 bits and this minimizes the required manipulation of data in order to store data in memory.

This block connects to the AHB Multiplexer and later to the AHB Slave, allowing the signals to finally reach the micro controller SubSystem.

7.8 MSS Routine

Once the data has been stored in the SRAMs, the micro controller Subsystem forwards it to the Ethernet output. For this to happen, a constant routine is repeated in an infinite loop to check whenever there is new data ready to be displayed. The system also needs the routine to let it know that the SRAM is ready to be updated from the logic side. The flow diagram that represents the MSS routine can be observed in Figure 7.6. Here, one can see that the routine is divided into a few more subroutines.

As a first step, the MSS routine compares the general information obtained from the traffic interceptor and if any change is detected from the previous data, it is sent out.

Later, using the control register, the microcontroller finds out if there is new data from any of the two available frames. These frames can be either complete or interrupted. Before sending the frame, the MSS reconstructs important parts of the frames mentioned before, so it can perform another function, which is reconstructing messages that are split between several frames. This allows the user to see full messages directly, and it helps when debugging, since it can be used to see if contiguous frames are being sent properly.

Once the frame data has been obtained and sent out, the control register is cleared, so the system knows it is ready to handle a new frame.

It is worth mentioning that to reduce time spent in this part of the system, the routine realizes raw memory copying from a static address that was described in the memory map, and end in the address that include Static Length of the Frame + Message Length + Process Data Length, that way, the routine does not need to handle the data since it is already pre-handled in the memory by the logic part, that way the process inside the micro controller subroutine its minimized.

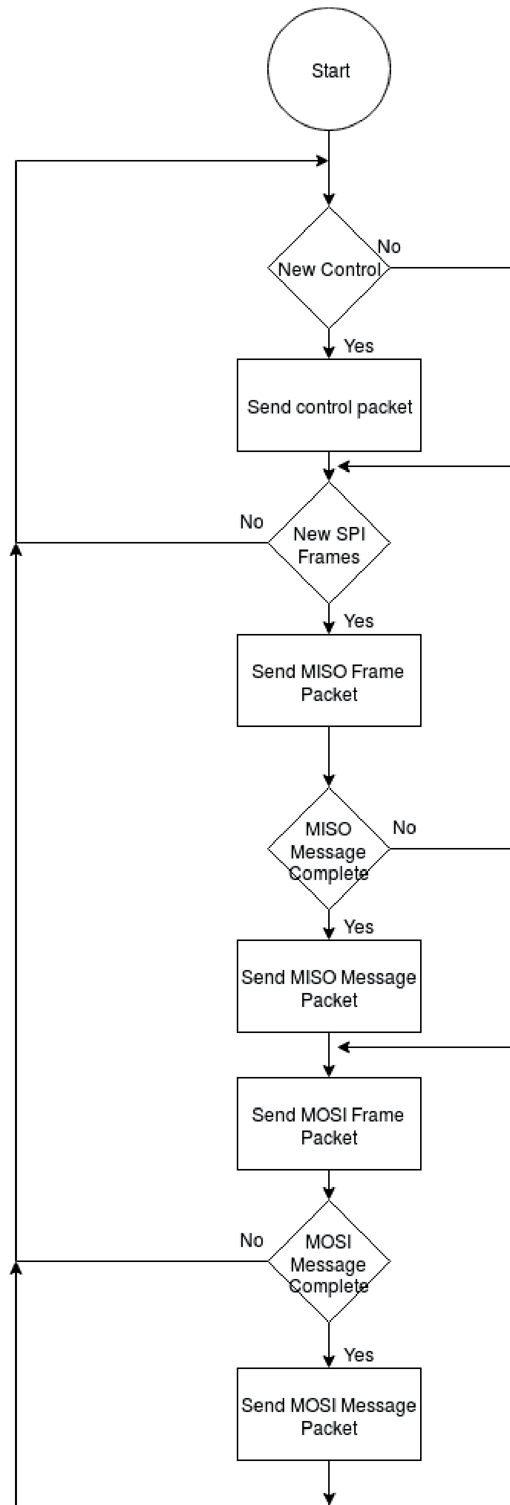

Figure 7.6: Micro controller Subsystem Flowchart

7.9 Wireshark Dissectors

As seen in the previous section, different data packets are sent to the Ethernet network, in order to be presented in a readable way. For that reason, the last component in the data chain is a network protocol analyzer.

Network analysis is the process of capturing and analyzing network traffic. It offers an insight into network communications to identify performance problems, analyze application behaviors and many more features.

At the moment, Wireshark is the world's most popular network analyzer, one of the reasons why it was selected is that it is an open source tool, that runs on a variety of platforms and offers the ideal 'first responder' tool for IT professionals. Another reason is that Wireshark is maintained by an active community of developers all over the world and it has extensive documentation available.

Due to the fact that Anybus is a protocol developed by HMS Industrial Networks and used for their products only, it is not included into the ones that Wireshark or any other network protocol analyzer can identify and ,for that reason, a set of dissectors were implemented by using the Wireshark standards. That way, the traffic intercepted by the interceptor can be successfully analysed.

For a frame to be recognized, it has to be recognizable by its corresponding dissector. To solve that, the MSS routine adds a byte in the beginning of the Ethernet stream that corresponds to the value the dissector is looking for. This is known in the Wireshark documentation as heuristics.

Dissectors work in a simple way. They take the desired byte stream and separate it in order to display it in a sorted way. A graphical representation of how a basic dissector works can be seen in Figure 7.7. In this Figure, the byte stream is divided into pieces and then the sorted the information is displayed in a readable way. The place where the data is displayed is called a tree. Moreover, Wireshark can handle bytes in a bit level, which is a useful feature for this implementation. There are 5 different dissectors available and they dissect the following byte streams:

- MOSI Raw Frame
- MISO Raw Frame
- MOSI Message reconstructed
- MISO Message reconstructed
- General Anybus information

Regarding the raw frames dissection, the representation of them is very similar to the one presented in Chapter 4. However, the MISO frame lacks the data regarding the message and process data length. To solve this issue, the MSS routine adds this data in the beginning of the byte stream. Once the byte streams

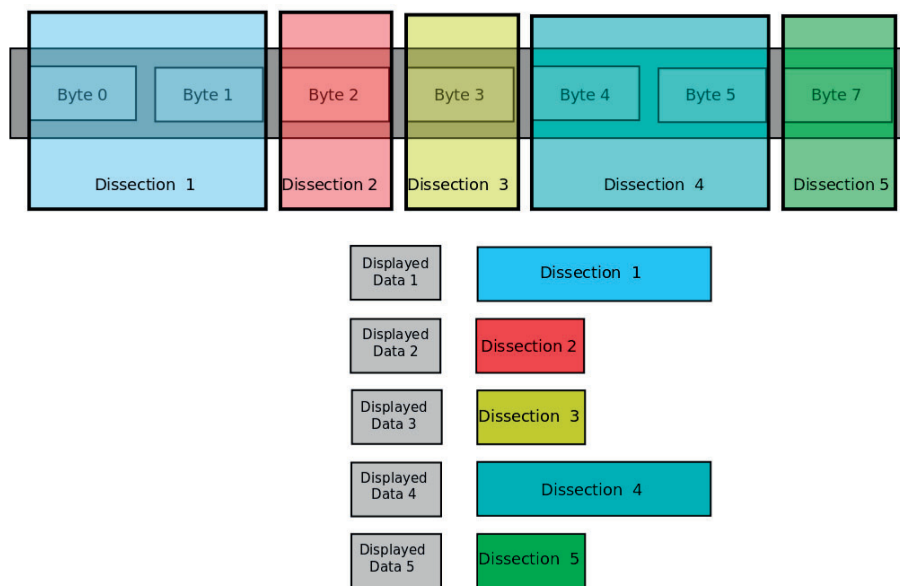

Figure 7.7: Graphical representation of how a Dissector displays data

have been captured, the dissector reads these values to be aware of the size that corresponds to that part of the tree, allowing it to be correctly displayed. The rest of the frame is static and can be always shown in the same way.

Byte	Item
0	Data field size - Low byte
1	Data field size - High byte
2	Reserved
3	Reserved
4	Source ID
5	Destination Object
6	Instance Low
7	Instance High
8	E C Command
9	Reserved
10	Comand External 0
11	Command External 1

Table 7.7: Message Header [6]

The reconstructed message dissectors display the message that has been reconstructed over the frames. However, these messages have a header, one for each frame, where important data about the message is stored. This data is used by higher levels of the software in the Anybus communication stack. Table 7.7 shows the data that in this header.

Lastly, the remaining dissector displays information regarding the general status of the Anybus CompactCom.

Chapter 8

Results

After assembling the hardware as described in Chapter 5.2 and implementing the traffic interceptor as explained in Chapter 7, the prototype was tested by generating traffic between a host CPU and the Anybus CompactCom. This traffic was generated by an application developed and provided by HMS Industrial Networks that emulated real data traffic.

Figure 8.1 shows a stream of messages sent in both directions. All of the messages in this figure have a message length of 8 and a pd length of 0. This is a testcase often used by HMS, since it works as a basic sanity check and it is similar to some typical messages used in real applications.

```
windows_tp
Press (r) to run free. Press (s) to step
sTarget Reset
Target Reset
Release target Reset
ANB_STATUS: ABP_ANB_STATE_SETUP
MOSI :870008000000000000000000000000000000000000000000C72805780000
MISO :00000000006A4DF6A34000000000000000000000000000000000C87E4E29
MOSI :1F000800000000000000000001030100410003000000000000A3277EA80000
MISO :00000000006448E0037000000000000000000000000000000000A2A455F7
MOSI :87000800000000000000000001030100410003000000000008643BC9C0000
MISO :00000000001EF0BEA235010000000103010001000300000000000660C35EF
RSP MSG_DATA_FORMAT: 0
MOSI :1F0008000000000000000000020301004100040000000000EAE479CA0000
MISO :0000000000067CD2853A0100000001030100010003000000000000AB8AB34C
```

Figure 8.1: Test traffic example used in prototype.

Using this application, two basic features were tested. The first test checking of that the frames were being correctly rebuilt in the dissector.

Figure 8.2 shows the output generated by reconstructing a single MISO Frame. Note that the reconstruction of a MISO involves information coming from the MOSI frame in order to determine its length. Further analyzing this figure, it can be noticed that when 2 frames are sent, 3 frames are captured. The reason for this is that each pair of frames includes a message that is also recovered. Furthermore, in the lower part of the Wireshark GUI, the whole Ethernet raw message is observed.

Here, the first bytes are set by the Ethernet decoder in the hardware provided. However, starting from the byte with value 0x14, the Anybus frame data is sent. This byte represents the dissector heuristics mentioned in Chapter 7.9, and is what differentiates the frame from others. The message length value is also seen there, 0x08 in this case, which represents the number of words that form the message. As mentioned before this frame does not include process data, so the value asigned for it is 0. Finally, a prefix byte is added at the beginning of the frame, which has a value of 0x01 if the frame is sent incomplete in the traffic or 0x00 if the frame was sent completely. From this point on, the stream data represents the Anybus frame. Above the raw data, the reconstructed frame can be seen. In this part, all the frame is divided and presented in sections, making it easier for the user to see the information on it. This test show that one single frame is recovered correctly.

Figure 8.2: MISO Frame reconstructed from a single frame.

As mentioned before, the third received frame is a reconstruction of the message sent between the Master and Slave, which can be seen in Figure 8.3. In this picture, the decoded header is shown, which corresponds to Table7.7.

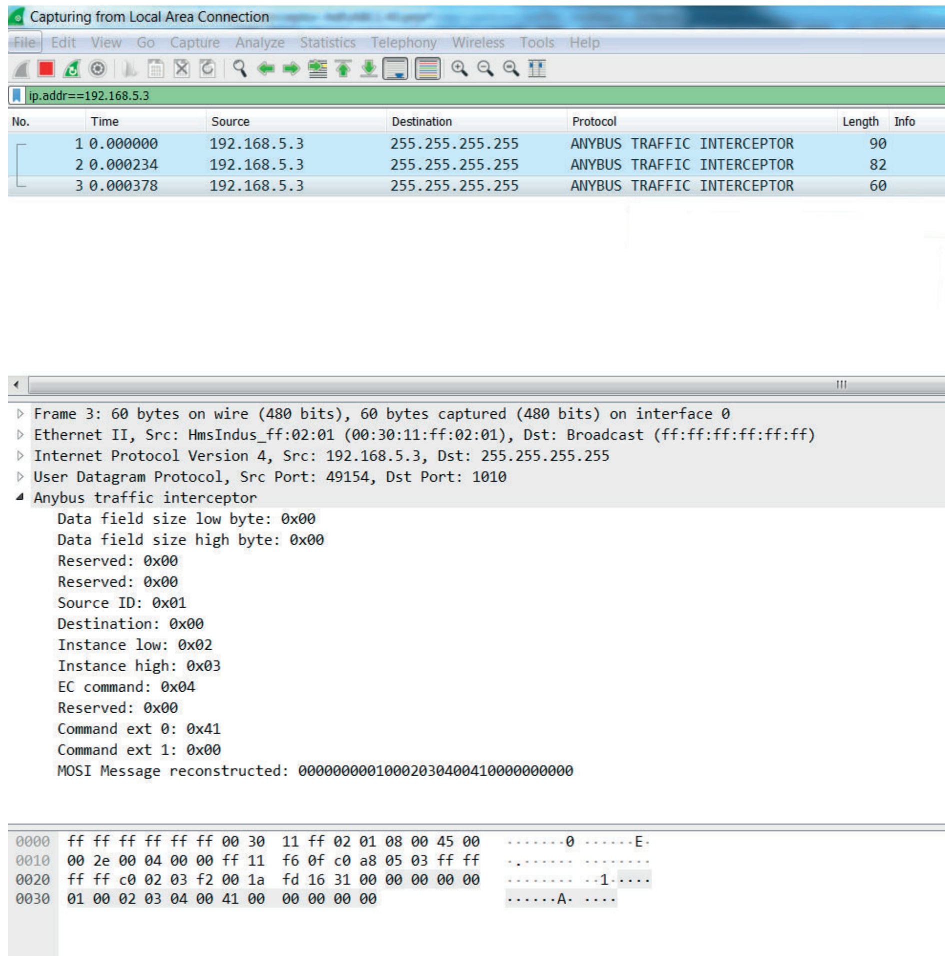

Figure 8.3: MOSI Message reconstructed from a single frame.

The second test that can be done by using this application is to sending a stream of frames, randomly selecting some of them and checking whether they are correctly reconstructed. This test is important since it allows us to check whether the traffic interceptor can capture more than one consecutive frame without dropping information. In Figure 8.4, one of the messages was reconstructed. It is worth mentioning that in this Figure, another feature developed in the dissector is shown: the possibility of refining the byte into bits to visualize what this byte in specific represents.

Figure 8.4: MISO Frame reconstructed from a stream of frames.

As much as this application is useful for basic testing, it is limited in the amount of different testcases and scenarios that it can emulate, since the frames have always the same length, which doesn't allow for proper streess testing of the traffic interceptor. For that reason, a test environment was developed, where FPGA logic of the traffic interceptor could be tested in a more flexible way.

8.1 Hardware Test

In order to test the traffic interceptor beyond the typical use case, a test fixture was implemented in a Xilinx Zynq Z7020 device. The Zynq family devices include standard FPGA fabric, referred to by Xilinx as Programmable Logic (PL), and a Dual-Core ARM Cortex-A9 processor, known as the Processing System (PS). Both parts of the chip are connected by a 32 bit AXI4 bus. This allows to feed frames from the microcontroller to the FPGA fabric, and format the signals in hardware as required by the interceptor.

The goal of this test is to emulate real traffic in different scenarios to characterize the traffic interceptor. In order to simplify the design of the testbench and

the analysis, only the FPGA side has been tested.

8.1.1 Hardware test architecture

The traffic interceptor test consists of two main parts: a traffic generator that can interface with the SPI side to emulate real traffic, and an AHB master that can be used to access the data in the interceptor, as the MSS would do in the full system. Fig 8.5 shows the different parts of the test hardware, as well as the connections to the traffic interceptor.

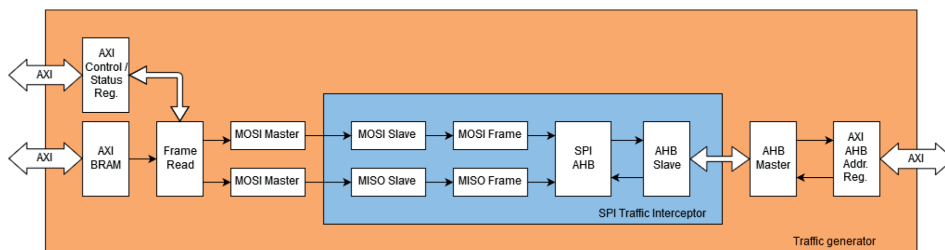

Figure 8.5: Traffic interceptor test hardware

AXI BRAM

This block implements a dual port 1 write 1 read RAM with a 32 bit width. This is used as the main way of communication between the Zynq CPU and the FPGA: the CPU writes a full frame to the BRAM, and the frame read block reads the data when necessary. Since an Anybus word is 16 bits wide, the CPU writes two concatenated words at a time, one each for MISO and MOSI frames.

AXI Registers

These are simple AXI registers which map to the CPU memory space and are used to control the test from the PS, as well as to send status data from the PL. All the registers have been configured to be unidirectional depending on their purpose: control and AHB address can only be written to from the CPU, while the status register is only written by the frame read block. This simplifies the design and avoids possible read/write collisions.

Frame read

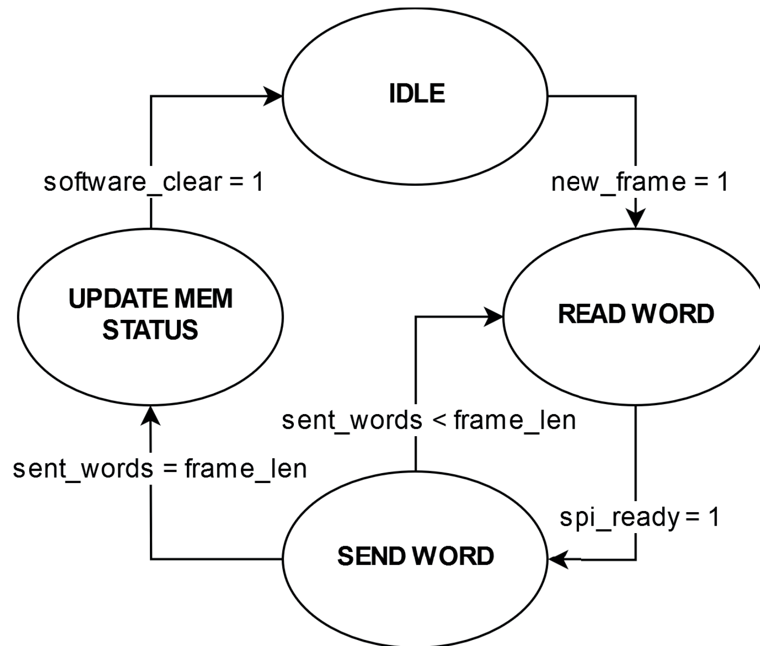

Figure 8.6: Traffic interceptor test hardware

This block is the core of the hardware test: it reads frame data from the BRAM and sends it to the SPI master blocks, while also generating the control signals for them.

The process of sending a full frame follows the state machine depicted in Figure 8.6:when the new frame bit in the status register goes high, the block reads a word from the BRAM and waits until spi ready is 1. This indicates that the SPI master is ready and the previous word has been sent. This process is repeated until all the words have been sent to the SPI, after which the block waits for the software clear signal. This is used to sync with the software.

SPI Master

This block implements the SPI master in both directions, which includes the MOSI ans MISO Master blocks shown in Figure 8.5. A counter is used to divide the FPGA clock signal down to the slower SPI clock. The generated SCLK signal runs at 25MHz, which is the fastest allowed SPI clock according to the Anybus specification.

AHB Master

This block interfaces with the AHB slave in the traffic interceptor. Since the reads in the test are all the time 32 bit, and there is only one slave, it is enough to have a

"dummy" block that takes the AHB address from the AXI AHB address register, and leaves the rest of the signals in the bus as constants.

8.1.2 Hardware test methodology

In the prototyping phase, the whole hardware and software stack was tested as shown in section 8. However, that test only tested messages with MSGLEN 8 and PDLEN 0. This is a typical use case for products using Anybus, but it doesn't cover other corner cases that could happen in a real implementation and would be legal according to the Anybus standard. It was therefore necessary to test more types of frames to gather performance data in different scenarios.

Test frames

In order to test the traffic interceptor in the FPGA, various frames where generated to be sent by the traffic generator. The main goal was to hit the extreme cases, which would be more prone to errors than usual scenarios. Because of that, most frames include combinations of either very large or very small MSGLEN and PDLEN parameters, which in turn create very long or very short frames, and some cases in between. The exact parameters are shown in Table 8.1.

Frame number	MSGLEN	PDLEN	Total length
1	0	0	6
2	8	0	14
3	8	2	16
4	1533	0	1539
5	1533	1533	3072
6	768	231	1005

Table 8.1: Test frame parameters. Lengths given in 16 bit words

Test procedure

The goal of the hardware test is to stress the traffic interceptor with traffic that emulates worst case scenarios that could arise in real applications. In order to do that, a testcase was created for each frame type, as well as an extra case with random frames. When running the testcase, the PS in the Zynq writes a frame to the BRAM and updates the AXI control register for the traffic generator to start sending traffic. When the frame has been sent, the status registers are updated by the traffic generator and the processor reads and stores the captured frame by updating the AHB address register. After sending all the frames, the captured frames are compared with the sent frames to detect possible errors, and the sent packet and correct packet numbers are shown via UART.

Testcase	Frame type	Sent frames	Received frames	Errors
1	1	30000	30000	0
2	2	30000	30000	0
3	3	30000	30000	0
4	4	30000	30000	0
5	5	30000	30000	0
6	6	30000	30000	0
7	Random	30000	30000	0

Table 8.2: Hardware test results

Table 8.2 shows the results for the different tests. No errors were detected for any size of frame or random frames. This indicates that the traffic interceptor is processing and storing the frame data as expected, and it remains stable for large amounts of traffic. The main bottleneck of the system is in the AHB communication between the FPGA and the CPU: since the FPGA is running faster than the SPI communication and it contains large enough buffers to store full Anybus frames, it shouldn't have any issue processing the incoming data, as shown here. The CPU then has to read that data before the next frame data overwrites the previous frame in memory. The "advantage" for the CPU is that the AHB bus is several times faster than the Anybus SPI, which gives it enough time to read, process and compare the data in the hardware test, or send it via Ethernet in the prototype. Another thing that helps give more time margin is the fact that the frames are not sent continuously, but with a $10\mu s$ space between frames as required by the Anybus specification explained in section 3.

8.2 Usage results

In terms of usage, the traffic interceptor utilization figures can be observed in Table 8.3. Here it is possible to notice that the design uses less than 20% of the available LUTs and 12% of the available Flip-Flops, which shows that the biggest part of the system relies on the Memories, of which 61% are used. In terms of I/O, the system usage is high due to the fact that the traffic analyzer has access to all the signals available. I.E Traffic bus, LEDS, etc.

Type	Used	Total	Percentage
4LUT	4514	27696	16.30
DFF	3331	27696	12.03
I/O Register	0	621	0.00
User I/0	134	207	64.73
RAM64K18	2	34	5.88
RAM1K18	19	31	61.29

Table 8.3: Anybus Traffic Interceptor Usage

8.3 Timing results

In terms of timing, Table 8.4 shows the clock frequency used, and the hold and setup values, proving that the design is timing clean.

Clock	Target freq. (MHz)	Period (ns)	Max. freq. (MHz)	Setup (ns)	Hold (ns)
HCLK	160	5.994	166.834	7.572	0.282

Table 8.4: Timing results

8.4 Power results

The power consumption that the interceptor requires can be observed in Table 8.5, where it can be seen that most of it corresponds to dynamic consumption. Total power consumption is in the order of a typical Anybus CompactCom, which is expected since the hardware is almost the same, and this means that the impact when used is negligible.

	Power (mW)	Percentage
Total Power	308.630	100
Static Power	17.409	5.6
Dynamic Power	291.222	94.4

Table 8.5: Power results

Chapter 9

Conclusions and Future work

It has been proved in this report that it is possible to build a traffic interceptor for the Anybus protocol using existing hardware and software as a base. This has been shown to be working in real hardware in typical and extreme conditions for the most common operation mode, the SPI mode.

It has also been shown, thanks to the implementation of a prototype, that this implementation can be used as a debugging tool for the Anybus traffic in real conditions, and, thanks to the hardware test, it has also been shown to work in extreme and worst case scenarios.

Possible improvements in the future could include implementation of the other operating modes, since there is still a lot of free space in the FPGA. Other future work includes higher level protocol decoding, such as Anybus objects, and accurate time-stamping for real time critical scenarios. This last feature could really benefit from the accurate timing and deterministic nature of the FPGA logic, allowing it to get better results than a software solution.

Appendix A

SmartFusion2 SoC FPGA Architecture

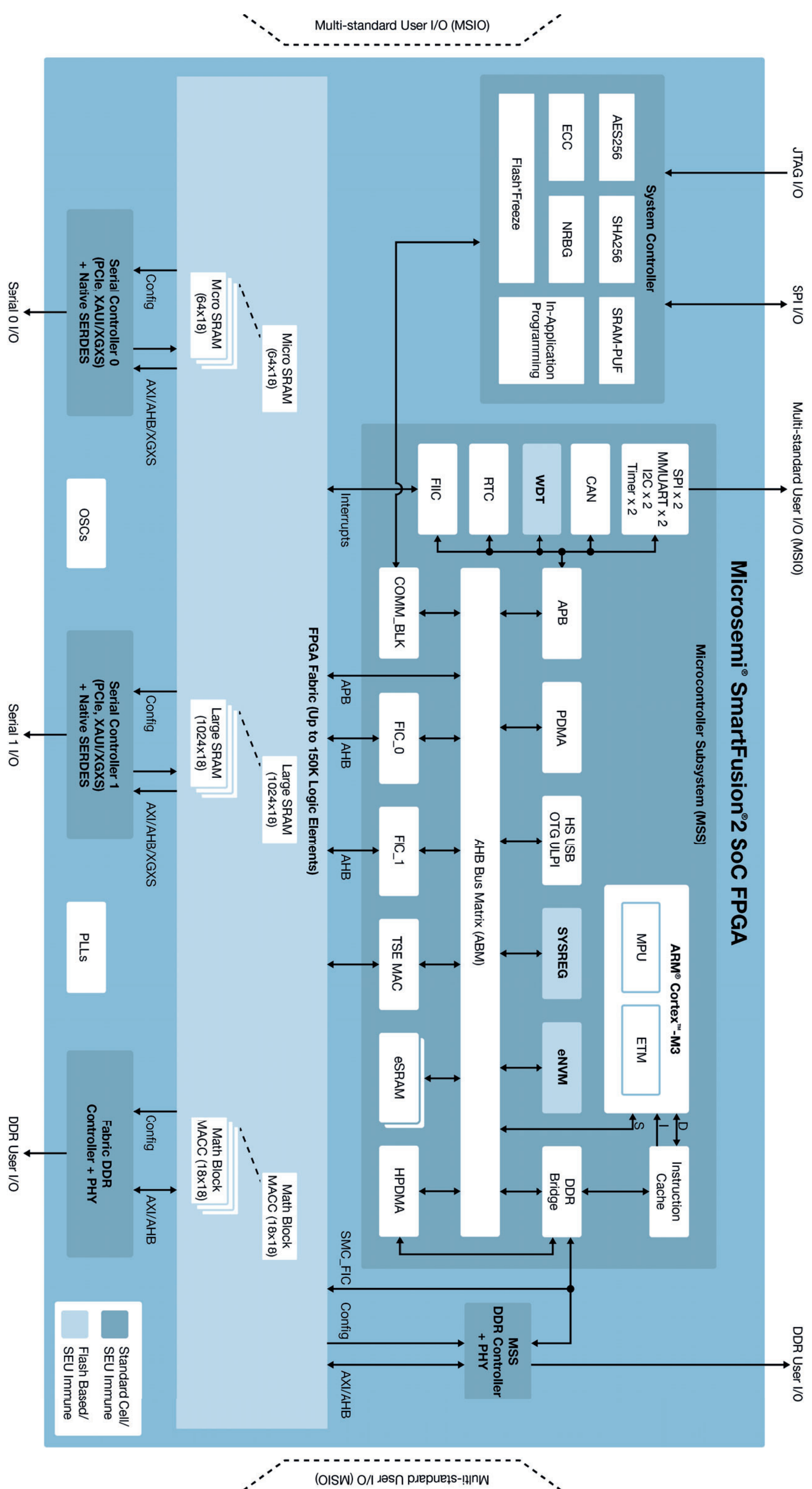

Figure A.1: Microsemi SmartFusion2 FPGA General Architecture.[4]

Bibliography

[1] HMS Industrial Networks AB. Anybus ®CompactCom ™ M40 Hardware Design Guide, 2017.

[2] HMS Industrial Networks AB. Anybus ®CompactCom ™ 40 Software Design Guide, 2017.

[3] Arm Limited. AMBA ®3 AHB-Lite Protocol AMBA 3 AHB-Lite Protocol Specification.

[4] Microsemi Corporation. DS0115 Datasheet SmartFusion2 Pin Descriptions.

[5] SPI Block Guide V03.06. 2000.

[6] HMS Industrial Networks AB. Case Study: Message Displays, 2015.

[7] João Faria and Arnaldo Oliveira. FPGA-based Ethernet Sniffer for Real-Time Networks. *Electrónica e Telecomunicações*, 2006(1):61–68, 2013.

[8] Chee Wei Liang and NNoohul Basheer Zain Miz Ramesh Seth Nair. Design of Low Cost FPGA Based PCI Bus Sniffer. Technical report.

[9] Sergiy Dorosh, Grzegorz Debita, and Patryk Schauer. Network hardware analyzer based on NetFPGA 1G. In *Proceedings - 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2017*, pages 150–154. Institute of Electrical and Electronics Engineers Inc., aug 2017.

[10] N Jayarathne and M K Jayananda. Development of a field programmable gate array based Controller Area Network sniffer. In *2013 IEEE 8th International Conference on Industrial and Information Systems*, pages 610–615, 2013.

[11] N Sriskanthan and Tan Su Lim. I2C bus analyser. *IEEE Transactions on Consumer Electronics*, 47(4):867–872, nov 2001.

Printed by Tryckeriet i E-huset, Lund 2020

LUND UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2020-752
http://www.eit.lth.se