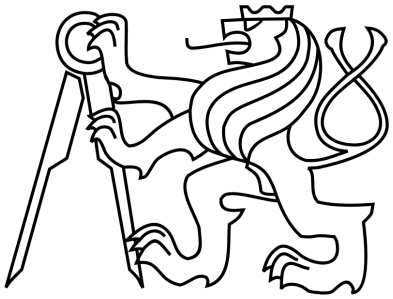
Czech Technical University in Prague

Faculty of Electrical Engineering

Department of Measurement



Master's thesis

Profibus Analyzer – Communication Driver and Firmware Update

Author: Bc. Daniel Kubeš

Supervisor: RNDr. Petr Štěpán, Ph.D.

Prague, May, 2018

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kubeš Daniel** Personal ID number: **420144**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Measurement**

Study program: **Cybernetics and Robotics**

Branch of study: **Sensors and Instrumentation**

II. Master's thesis details

Master's thesis title in English:

Profibus Analyzer - Communication Driver and Firmware Update

Master's thesis title in Czech:

Modifikace programového vybavení komunikačního zařízení Profibus Analyzer

Guidelines:

1. Familiarize with the Siemens internal project Profibus Analyzer and RT Linux kernel system.
2. Port U-Boot & Linux (based on version 4.9.28) to Profibus Analyzer board, which is based on TI SITARA MCU.
3. Design & Implement the Linux kernel driver for communication with FPGA based on implemented parallel interface, adapt the existing application of Profibus Analyzer if necessary.
4. Design & Implement the firmware update strategy for Profibus Analyzer which includes the FPGA firmware, Linux kernel driver and Application.
5. Verify and document the solution.

Bibliography / sources:

Name and workplace of master's thesis supervisor:

RNDr. Petr Štěpán, Ph.D., Multi-robot Systems, FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **09.10.2017** Deadline for master's thesis submission: **25.05.2018**

Assignment valid until:
by the end of summer semester 2018/2019

RNDr. Petr Štěpán, Ph.D. Supervisor's signature	Head of department's signature	prof. Ing. Pavel Ripka, CSc. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Author statement for undergraduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date: ..... Signature: .....

Acknowledgement

I would like to express my gratitude to my supervisor RNDr. Petr Štěpán, Ph.D. for his supportive guidance, his useful advice and comments during the writing.

I would also like to thank my family and my girlfriend who supported me during the whole time of my studies.

Last but not least, I would like to thank my colleagues from the Siemens for the support with the practical part of the thesis.

Abstrakt

Tato diplomová práce se zabývá vývojem firmwaru pro zařízení Profibus Analyzer od firmy Siemens. Zařízení bude možné využít pro analýzu dvou kanálů Profibus DP a jednoho kanálu PA v jedné časové doméně, což je nezbytně nutné pro precizní analýzu Profibus zařízení, které využívá redundatní komunikační kanál.

V práci je popsán vývoj komunikačního řetězce sloužícího k přenosu dat z FPGA, které zpracovává data z Profibus kanálů, do PC aplikace. Jeho základem je procesor Sitara, na kterém běží operační systém Linux. Zmíněný komunikační řetězec obsahuje modul linuxového jádra, který zajišťuje přenos dat z FPGA do paměti procesoru, real-time aplikaci, která komunikuje s kontrolní PC aplikací a předává ji naměřená data z FPGA a nakonec dynamickou knihovnu (.dll) pro zapouzdření komunikace mezi Profibus Analyzerem a počítačem, která bude využita pro kontrolní GUI aplikaci.

Klíčová slova

Profibus, Profibus DP, Profibus PA, Fieldbus, Analyzátor, FPGA, Sitara procesor, AM335x, TI, Linux, RT patch, Kernel modul, Real Time aplikace, Server-klient, Siemens

Abstract

This diploma thesis focuses on the firmware development for the Profibus Analyzer device made by the Siemens company. It will be possible to use this device to analyze two Profibus DP channels and one PA channel in the same time domain, which is necessary to achieve a precise analysis of the Profibus device, which uses a redundant communication channel.

The thesis describes the development of the communication stack, which ensures the data transfer from the FPGA (that processes the Profibus channels data) to the PC application. The communication stack is based on the Sitara processor running the Linux OS. This stack contains Linux kernel module, the real-time application and the Dynamic Link Library. Linux kernel module is responsible for the data transfer from the FPGA to the processor memory, the real-time application communicates with the control PC application and forwards the FPGA data to the PC and the Dynamic Link Library encapsulates the whole communication between the PC and the Profibus Analyzer to be used by the control GUI application.

Keywords

Profibus, Profibus DP, Profibus PA, Fieldbus, Analyzer, FPGA, Sitara processor, AM335x, TI, Linux, RT patch, Kernel module, Real Time application, Server-client, Siemens

Contents

1 Introduction

Nowadays, when a manufacturing process becomes more automated, new technologies are introduced almost on a daily basis and a human becomes an assistant of a robot, the Industry 4.0 concept was introduced (the name was first used in 2011 at the Hannover Fair). The key idea is to make communication among whole range of devices possible so they could cooperate more effectively - from the industrial machines, actuators and sensors to the IoT (Internet of Things). This means that a complex computer networks have to be used in factories. One family of these networks is called Fieldbus (standardized as IEC 61158).

Fieldbus finds its place everywhere in the factory, where instruments (such as PLCs to sensors or actuators) have to be connected together and perform real-time communication. An old version of Fieldbus standard contained eight different protocol sets (called "types"), but the current version from 2008 reorganized types into Communication Profile Families (CPFs). [1]

PROFIBUS (Process Field Bus) is one of CPFs and therefore also a family member of the Fieldbus standard. It is used in many applications in a field of factory automation and in a safety applications like chemical plants or power stations as well. For the safety applications, redundancy concepts were introduced by specification (only vendor-specific redundancy concepts were available earlier) to achieve required higher safety level. There are few kinds of redundancy defined in Profibus - Master redundancy (the control system is redundant), Media redundancy (uses redundant media for transmission), Coupler/gateway redundancy (where if one fails, the other one will take care about coupling), Ring redundancy (which is a combination of few redundancies and therefore more complex) and a Slave redundancy (where field devices are redundant). [2]

For every communication technology it is necessary to provide some diagnostic tools for analysis, data capture and statistics measurement. If we take a look at a situation on a market, there are few solutions for Profibus channel diagnostic - for example Procentec Analyzer or Softing Analyzer. They share some common features such as USB interface, a scan of topology or bar graphs of measured data and both devices provide PC application. Procentec provides support for Profibus DP or PA (one at a time) and Softing provides support for both Profibus DP and PA at the same time. But these solutions are not sufficient enough in the case when any redundancy feature of Profibus is used. There is actually no suitable solution for this problem at all, which would be able to analyze Profibus Slave Redundancy (2x DP channel) and a PA channel in one time domain. [4]

A new solution for Profibus analysis that will be able to analyze not just a single Profibus channel, but also applications of slave redundancy in one time domain is currently being developed by Corporate Technology department of Siemens Company. This solution will be called a Profibus Analyzer in this paper.

The aim of this thesis is to develop a firmware part of Profibus Analyzer running on the Sitara processor (which includes the kernel platform driver and the Linux application) and to design a communication protocol for control application running on PC. However, this thesis does not contain the development of hardware and FPGA application, which have been already prepared. The control PC application (GUI) is also not contained in this thesis. It is important to distinguish between two terms, which will occur in the thesis - Profibus Analyzer Application is a firmware part of the thesis, however control PC application is not a part of the thesis, but it will be mentioned several times also.

2 Profibus

In this chapter, only the main features Profibus technology are mentioned, because it is impossible to put the whole description of this technology into this work.

As mentioned in the Section 1, Profibus (Process Field Bus) is a part of the Fieldbus standard and it is widely used in industrial automation system networks.

2.1 History

Profibus was developed in Germany in the late 1980s by several companies such as Bosch, Siemens and Klöckner & Möller in cooperation with German universities.

The 1990s were quite an interesting historical period from the technological point of view. It was called The German-French Fieldbus War. The experts realized that the ensuring of compatibility would become too difficult with many different protocols, so they finally decided to create one standard to cover all used protocols. They tried to combine German Profibus, French FIP (Factory Instrumentation Protocol) and subsequently another ones, but the approaches of these systems were different and it took a lot of effort to combine them in the best possible way and a lot of compromises had to be made. During the process (which took more than 8 years), the attempt to combine these technologies was called sarcastically "The two-headed monster" due to different approaches. First success, and for the long time the last one, came in the 1993, when the physical layer definition was released by IEC 61158-2 and has been widely used since. [2]

Above the physical layer, the standardization did not work properly. The result was always too complex, complicated to maintain, too expensive and therefore almost useless. Because the attempts took so much time, Fieldbus systems were already on the market and were already used in industrial automation and nobody wanted to abandon his product because of the new standard. Therefore a completely new way was introduced - not to combine the Fieldbus systems to a single technology, but to make a standard accommodating all Fieldbus systems as it was. The compromise was finally made in 1999 when standard (IEC 61158) accomodating eight Fieldbus systems was released by Fieldbus Foundation. [2]

2.2 Present time

Nowadays, Profibus is no longer used for new projects in such amount as at the turn of the millennium, because more advanced technologies such as Profinet were developed. Nevertheless, there are still millions of devices, which have to be maintained and supported for the whole lifecycle. Moreover, it is used also in projects where the potential redesign to more advanced networks would cost a lot more money and also time than preserving a Profibus communication (for example systems in a nuclear power plant). For that reasons new devices are still developed and Profibus is still well maintained.

2.3 Topology

Profibus uses a bus topology. In case of a multi-master network, the Token passing method is used for media access control - a token is passed among all master devices (active nodes) in a ring - only a master that is in possession of the token can transmit. The other masters have to wait until they get the token to be able to start the communication and the slave devices (passive node) have to wait till the master with a token starts the communication with them (by polling). Just a simple master/slave communication scheme is used in the case of a single master. [3]

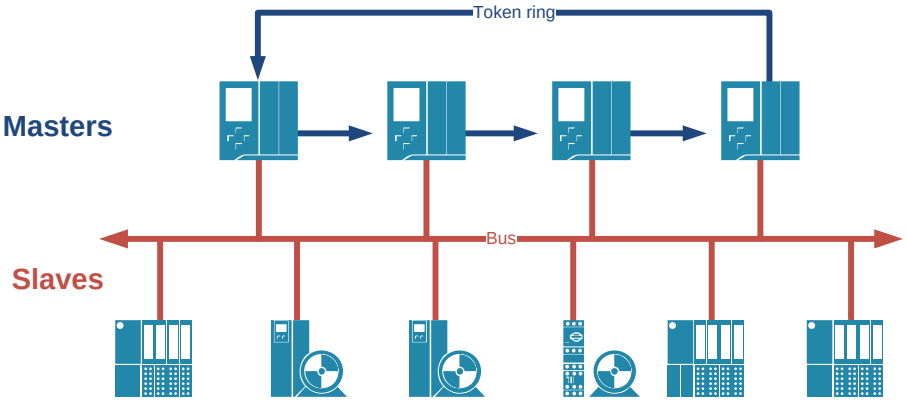


Figure 1: Bus topology

2.4 ISO/OSI

ISO/OSI reference model is a result of attempts to standardize the computer networks - it divides the communication procedure into seven different layers, which solve different kinds of problems of communication procedure.

Profibus defines just three layers of ISO/OSI reference model - Application layer, Data Link layer and Physical layer. The other four layers are not used by the Fieldbus standard. [3]

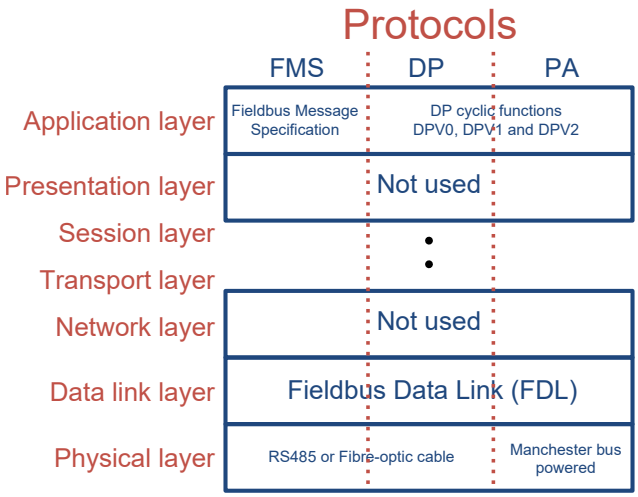


Figure 2: Profibus ISO/OSI model

2.4.1 Application layer

This layer provides services for final applications and hides the layers below for the user. To be able to work in so many different areas with a different desired performance, the Application layer provides three performance levels of DP cyclic functions:

- DP-V0 provides the basic functionality, which includes the cyclic communication and some part of diagnostics (for example indication of short-circuited output).
- DP-V1 is an extension of the previous DP-V0. The acyclic communication is added, which enables the use of alarms, parametrization and online access to the bus using debugging tools.

- DP-V2 is an extension of the previous DP-V1 (and therefore contains also DP-V0 functionality). It allows slaves to communicate with another slave, cycle synchronization and time stamping.

Another component of the Application layer is the Profibus Fieldbus Message Specification. It is the initial form of Profibus protocol that provides complex message structure. It is used for communication between two masters acyclically (peer-to-peer), as it was intended to allow coordination between different PLCs. Nowadays it seems to be fading away, because the Profibus DP in combination with FDL can replace this old feature. [2]

2.4.2 Link layer

This layer is called Fieldbus Data Link (FDL) layer in Profibus standard. It is responsible for media access control, which is described above in the Subsection 2.3.

The most important point is that this layer, this media access method in more detail, makes Profibus a deterministic network and therefore it can guarantee that every single device will be accessed within a fixed time.

FDL is also responsible for the autodetecting of a bus speed which is slave connected to (so the speed of the slave is configured automatically). Another feature of FDL is that it provides a network scan - FDL request is periodically sent by the master to ensure that the slave is still present on the bus and is able to respond. To ensure at least a minimal level of security, even parity bit is added to each byte with a start and a stop bit by FDL.

2.4.3 Physical layer

The Physical layer, how the name tells us, takes care of the physical transfer of information to another device bit by bit. Profibus standard provides a few different ways how the transmission technology can be implemented:

- Metallic cable is based on a well-known standard RS-485. It uses a twisted copper cable with a pair of conductors to avoid interferences in noisy environments (which is not a rare situation in the industry). A logical level is determined by the voltage difference between the pair of conductors, therefore devices can have a different power supply with different ground offsets. The maximal data transmission rate is 12 Mbps. The speed is determined by network complexity and a cable length, which can range from 100 m to 1 km (without repetitors). Metallic cable is the most common Physical layer.
- Fiber-optic cable is even better in noisy environments than RS-485 variant, because no electromagnetic noise can interfere with transmission in the optic cable. High transmission speeds can be reached for larger distances.
- Intrinsically safe transmission is defined for Profibus PA (described below). It is a standard for hazardous environments, which uses a current loop (industry standard of 4-20 mA). It has the fixed data rate at 31.25 kbps.

2.5 Protocols

There are several versions of Profibus protocols - Fieldbus Message Specification (FMS), Decentralized Peripherals (DP) and Process Automation (PA). Protocols are defined by a complete set of ISO/OSI layers features as indicated in the figure 2.

2.5.1 Profibus DP

Profibus Decentralized Peripherals is used when a central controller wants to communicate with actuators or sensors, but it can also be used to communicate with other controller and therefore makes internetworking of several controllers possible (multi-master devices).

This protocol provides both cyclical and acyclical communication and defines rules for them. The Application layer uses DP cyclic functions, the Data Link layer uses FDL and the Physical layer uses metalic or fibre-optic cable.[14]

2.5.2 ProfibusPA

Profibus Process Automation has almost the same features as DP except the Physical layer, where the intrinsically safe transmission is used. It can be used for the control measuring devices in process engineering and it is suitable for explosion or hazardous areas. Another feature is that it supplies the end devices via bus cable. [3]

PA network always has to be a part of a DP network, because it cannot exist without a DP master. [3]

2.6 Telegrams

The data and messages are transferred in the Data Link layer using the Profibus telegrams. [14]

2.6.1 Transfer type

There are two ways of data transmission defined by the Link layer - SRD (Send and Request Data with acknowledge) and SDN (Send Data with No acknowledge) for Profibus DP.

As the names tell us, if SRD type is used, the device sends data and at the same cycle it expects some response with the data. This way is therefore really effective and fast, because the communication is performed just in one telegram cycle.

The SDN type is used whenever the message should be sent to the group of devices or to all of the devices (broadcast). The acknowledge is not expected on the sender side.

Just note here that there is another type called SDA (Send Data with Acknowledge) that is used for Profibus TMS. The acknowledge is expected to be received by the sender.

2.6.2 Characters

The telegrams are created by characters - the following description applies to every byte transfer over Profibus.

Each character length is strictly fixed to 11 bits, which contains a single byte of data, start bit, stop bit and parity (even). Bits are transferred using the NRZ (Non-Return to Zero) code with LSB (Least Significant Bit) first (in case of transfer of word it is used Big-Endian format). The idle state is logic one, when the start bit is transmitted, it pulls the line down. This format of transfer is known for example from the RS232 standard.

2.6.3 Telegram structure

The telegram contains several parts (or fields), all of them can be found in the list below according to the position in the telegram, from the beginning to the end (the size of each field in bytes can be found in brackets):

- Start Delimiter (1B) - beginning of a telegram, it determines the basic type of telegram (request, response with variable data length, response with fixed data length, token and no SD).
- Data Length (1B) - the number of bytes between DA (Destination Address) and the end of DU (Data Units). Usually, the DU size is up to 32 bytes although the protocol allows to use up to 244 bytes.
- Repeated Data Length (1B) - Data Length field repeated for protection.
- Destination Address (1B) - valid addresses are in range of 0-127 (0x0 - 0x7F). Special cases are the address 126, which is reserved for commisioning and the address 127, which is reserved for broadcast. The most significant bit is used as an indicator for the inclusion of the DSAP and SSAP (see below).
- Source Address (1B) - the same situation as DA except the address of 127. When a message is being broadcasted, there is no response and therefore the source address cannot be 127.
- Function Code (1B) - it specifies the type of telegram, type of station, priority and telegram acknowledgment.
- Destination Service Access Point (1B) - it specifies which services should be performed (for example read inputs, read outputs, read diagnostics, check configuration, etc.). It is used only if the MSB of Address is set.
- Source Service Access Point (1B) - the same situation as DSAP applies here.
- Data Units (1B - 244B) - data to be sent to the device on the Destination Address.
- Frame Checking Sequence (1B) - the checksum of the whole telegram.
- End Delimiter (1B) - the fixed value of 0x16 indicates End Delimiter and therefore also the end of the frame. [14]

The communication using SRD transfer type is captured in the figure 3.

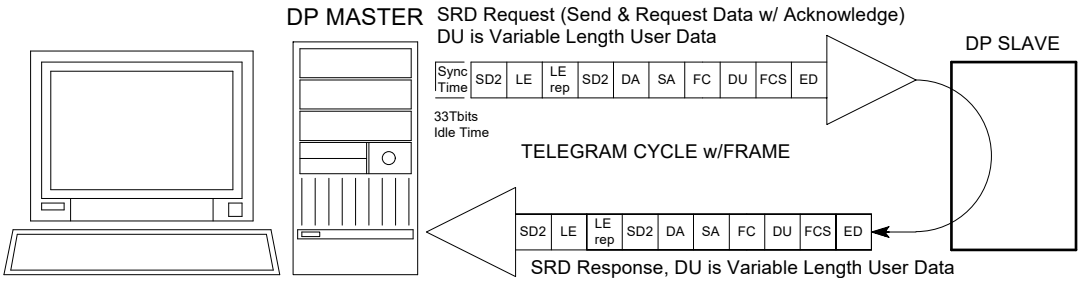


Figure 3: Example of Profibus communication [14]

3 Profibus Analyzer device overview

Profibus Analyzer will be an internal device of the Siemens company, which is currently being developed in Corporate Technology (R&D) department in Prague.

The list of planned properties:

- Input for two Profibus DP channels (9,6 kbps – 12 Mbps).

- Input for single Profibus PA channel (31,25 kbps).
- Three input triggers.
- Three output triggers.
- Ethernet interface.

3.1 Hardware

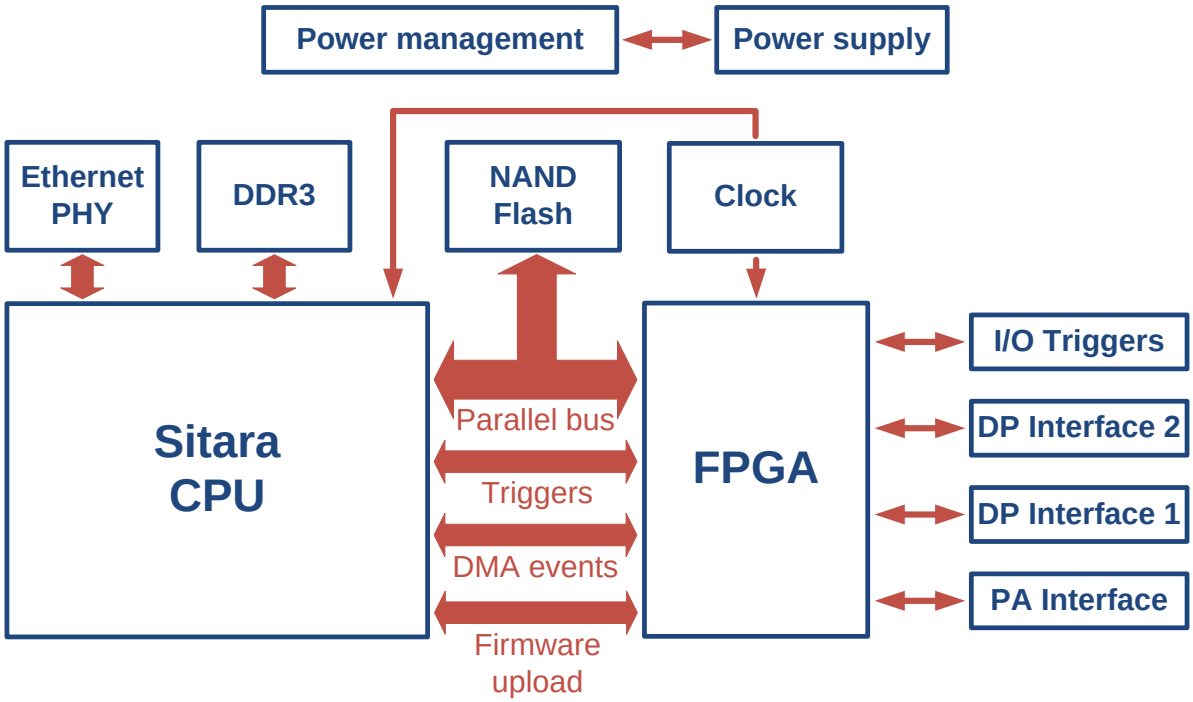


Figure 4: Block schematic

The core of the hardware is a Sitara CPU made by Texas Instruments, which is connected to the FPGA Cyclone IV. Profibus channels are connected via isolators to the FPGA, which will take care of adding timestamps and then of message forwarding to the CPU via the parallel communication interface. Finally, the communication between the CPU and the control PC running the control software is performed via the Ethernet interface.

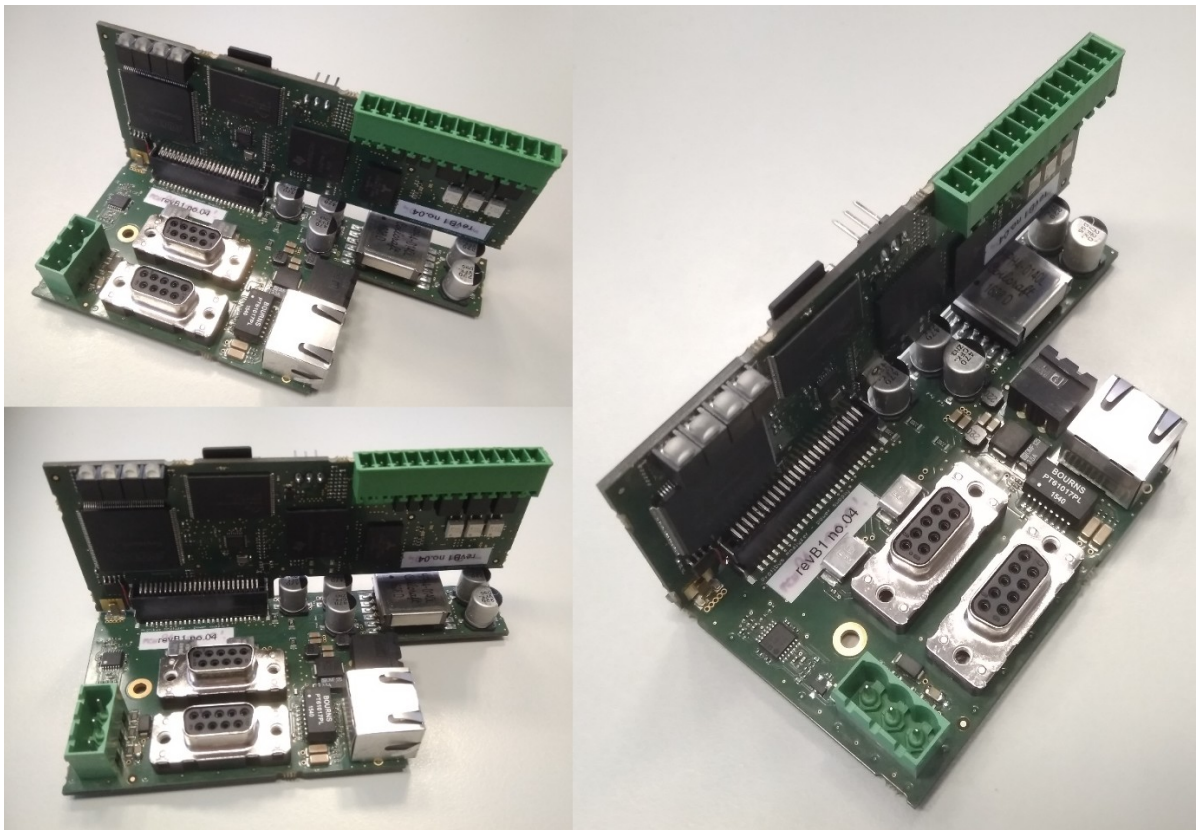


Figure 5: Profibus Analyzer hardware

3.1.1 Sitara processor

The CPU AM3352 (that belongs to Sitara Processors family) was chosen to improve the performance of the board. It has all necessary features including the DMA, the memory controller and the higher clock frequency (up to 800 MHz). [10]

The list of main features of CPU AM3352 used in this work:

- GPMC - General Purpose Memory Controller is used to access the FPGA, which is connected to the address and the data bus of the CPU and acts as a dynamic memory. The parallel memory interface is used so the communication bandwidth between the CPU and the FPGA is increased significantly (compared to the previous version of Profibus Analyzer, which used 2x SPI, the memory interface should be 8 times faster).
- EDMA - Enhanced Direct Memory Access saves time of the CPU, because it takes care about memory transfers with just a minimal effort of the computing unit. In our case, it transfers data from the FPGA (via GPMC) to the DDR3 automatically using DMA transfer trigger pins that are used by the FPGA to trigger the DMA transfer whenever the data are ready at the FPGA side.
- EMIF - External Memory Interface is used to connect the DDR3 memory to the CPU. The DDR3 is used to store runtime data and Profibus telegrams. For storing static data as Linux filesystem and kernel, the flash memory will be also connected.
- SPI - Serial Peripheral Interface is used to load the bitstream to the FPGA using the Passive Serial Programming mode (it will be described later in the Section 7.3).
- Ethernet MAC - Ethernet is used to send data to the control PC.

- UART - Universal Asynchronous Receiver-Transmitter is used for the Linux console to be accessible using the PC terminal.
- GPIO - General Purpose Input Output pins are used for several purposes - for changing the LED state (for example the heartbeat of the Linux kernel) and to control signals for the bitstream loading to the FPGA.

3.2 Software

The software part of the Profibus Analyzer can be divided into four categories:

- Operating system - Standard RT-Linux kernel for Sitara processor, which is distributed by Texas Instruments company in version V4.9.28 will be used. Board-specific and Core-specific files of kernel will be rewritten to be compatible with the Profibus Analyzer hardware. The Profibus driver is added as a platform driver.
- Profibus kernel driver - FPGA designed for receiving bytes from the Profibus network adds timestamps and sends internal telegrams, which contain received Profibus telegrams, to the CPU.
- Profibus Linux application - The POSIX-compliant application, which is cooperating with kernel drivers via standard read/write mechanisms. The base behavior of the application is the interconnection of the Profibus driver and the TCP/IP stack to transfer all captured telegrams to the PC application.
- PC application - GUI application that displays information about all telegrams received from the Sitara in a clear, readable and user-friendly GUI.

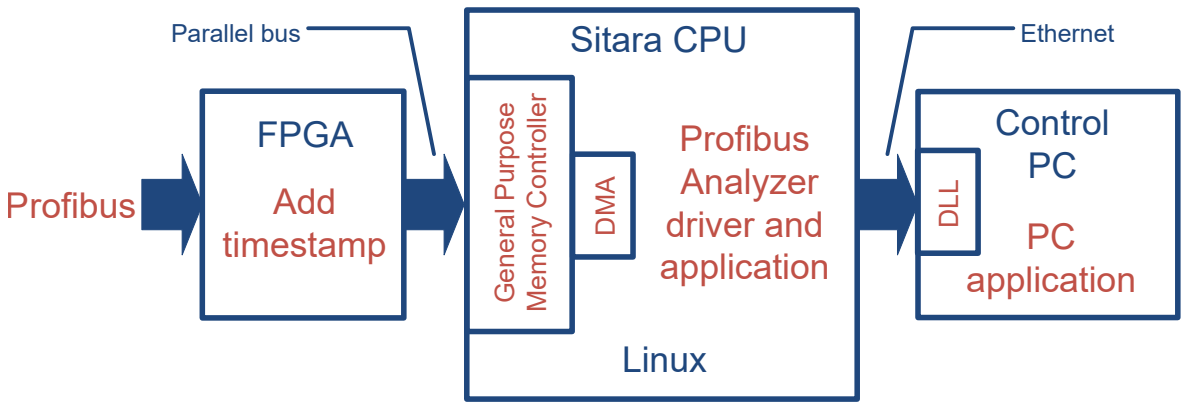


Figure 6: Software diagram

In the figure 6 the connection between separated software parts is clearly presented. Every piece of software is being developed separately.

3.3 Profibus Analyzer use cases

In this section, the use cases of Profibus Analyzer are presented. As mentioned in the Section 1, the main goal is the analysis of Slave redundant Profibus, but there are of course more use cases than this. All cases that are listed below can be extended by using input/output triggers. Using the input triggers, the measurement can start at the proper moment - for example exactly at the same time when the reset signal to the device is deactivated. Output triggers can be used for example to assert

the reset signal of the device or, in case it is connected to the interrupt input pin of the device, it can cause the interrupt.

List of standard use cases:

- The analysis of functionality of Profibus devices such as repeaters. Profibus Analyzer can be connected to the line in front of the repeater and to the line behind the repeater. Therefore it can be analyzed if the repeater works as expected.
- Replacing the repeater from the previous use case with another device, which has some internal logic, the delay between incoming and outcoming telegrams can be precisely computed.

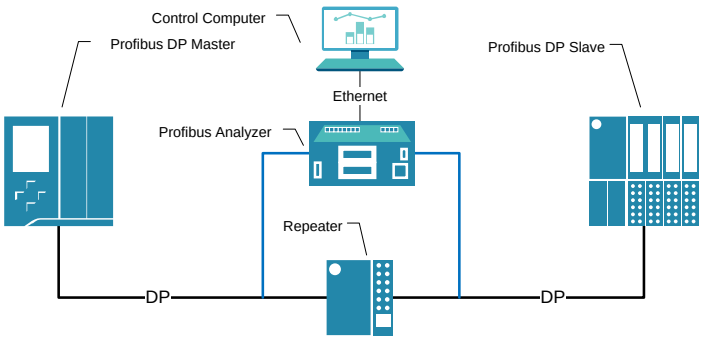


Figure 7: Use case: repeater analysis

- It is possible to analyze two different Profibus networks using a single device. The other benefit is that we get the timing information so we can analyze the synchronicity of these networks.

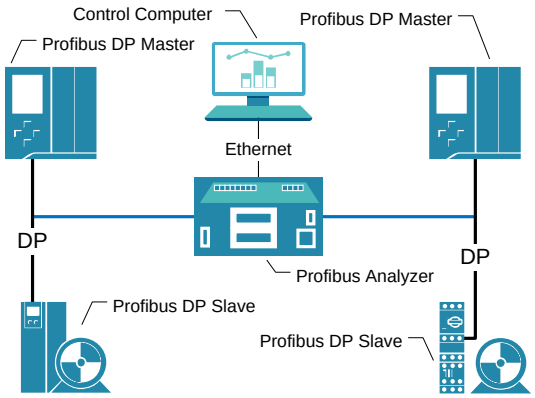


Figure 8: Use case: simultaneous analysis of two networks

- Profibus Analyzer makes it possible to analyse the whole redundant communication in one time domain, which is the main feature and literally a reason, why the development was initialized. Differences in timing can be easily find out, so the debugging of such a network will be much more comfortable. The analysis of redundant communication using two independent devices is a challenging task and it is almost impossible to get really reliable results without our device.

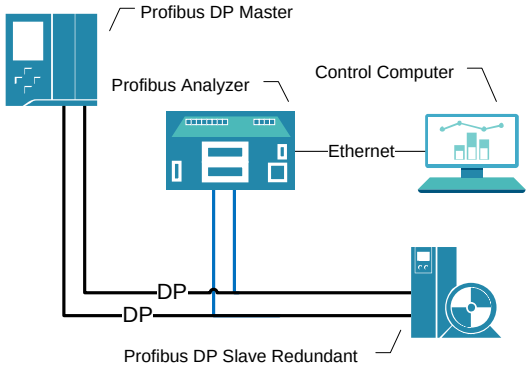


Figure 9: Use case: redundant communication

- It is possible to analyze not just the whole redundant communication, but also another PA channel in one time domain. It uses all of the mentioned benefits from the previous case, but moreover the additional PA channel can be analyzed there and therefore for example the delay of a device, which receives data from redundant DP channel and then transmit another data via PA channel, can be estimated.
- Because the DP and PA channels can be analyzed simultaneously, it is also possible to analyze the functionality of DP/PA coupler device (a device that creates a bridge between DP and PA channels and makes them compatible).

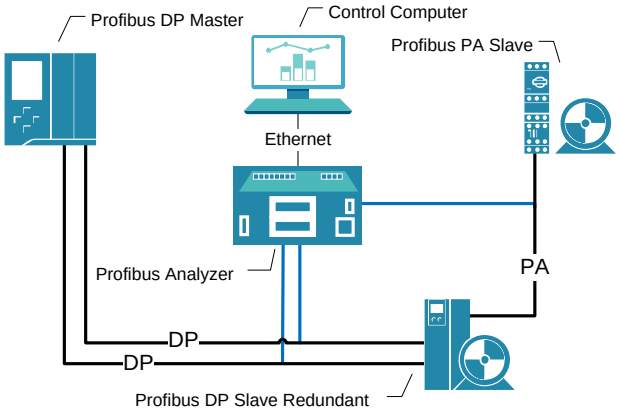


Figure 10: Use case: redundant communication with PA channel

4 Linux

In this chapter, the basics about Linux and Linux kernel are described. There is also a short history overview, the introduction to the Real Time systems and to the Real Time patch respectively.

4.1 Introduction

Linux is an open source Unix-like operating system based on Linux kernel developed under a license of GPLv2.

The most important features of Linux [5]:

- Unix-like type of operating system, which for example means that it is, or should be, kept as simple as possible and basic configurations and system files are saved as plain text (ASCII) files.
- It provides the support for
  - multiprocessing
  - multiprocessor architectures (wide support for different processors from different manufacturer)
  - 32/64 bit architectures
  - multi-user environment
- Preemptive OS - the process cannot block processor for itself.
- Quickly spreading to all areas - from the classical personal computers to the embedded systems.
- A large number of developers and therefore good support.

4.2 History

The first version of Linux kernel was developed by Finnish developer Linus Torvalds in 1991. The operating system was published on 17th September and Torvalds immediately started to get a lot of responses by mail, so he decided to keep developing the source code and publish new versions. During the next years, thousands of developers have started to work with Linux and therefore it spread all over the world. Originally, the Linux kernel was written for the i386 processor architecture, but during the time it was ported to many other platforms such as the ARM, the MIPS, the Atmel and so on.

The name Linux was chosen by the administrator of the FTP server, where the first version was published. Linus Torvalds wanted to name his operating system "Freax" as a connection of words free, freak and Unix, but the administrator Ari Lemmke did not like it, so he chose Linux as a connection of the name Linus and Unix. [1]

In the figure 11, there is a logo of Linux - a penguin called Tux. Tux was created by American programmer Larry Edwing in 1996. [6]

[1] Just note here, that Linus did not like this version, because it seems too self-centered to him.

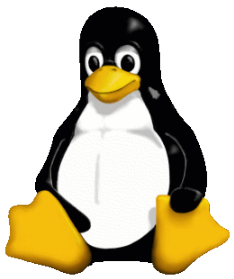


Figure 11: Linux logo

4.3 Kernel architecture

The Linux kernel is a monolithic Unix-like operating system kernel which supports the loading of extensions (kernel modules). This architecture was chosen to keep the kernel more stable and maintainable. Linux operating systems are based on the Linux kernel and extended by custom modules.

The Linux kernel supports the preemptive multitasking (more application can run there simultaneously), the virtual memory (paging, each process has its own memory space), the memory management (the memory is managed only by the kernel, not by userspace applications), kernel modules/drivers and networking. Kernel modules (usually for new hardware support) can be dynamically loaded/unloaded from the kernel while the system is running.

There are two separated parts of the virtual memory (which define different levels/protection rings of privileges) - the kernel space (ring 0) and the userspace. This method provides the memory protection - the user space application (the application software and some drivers) cannot make inappropriate access to the memory (only to its own memory space) so it cannot corrupt any other process. On the other hand, from the kernel space, which is strictly reserved for the kernel, any memory is accessible, including hardware peripheral registers. [5]

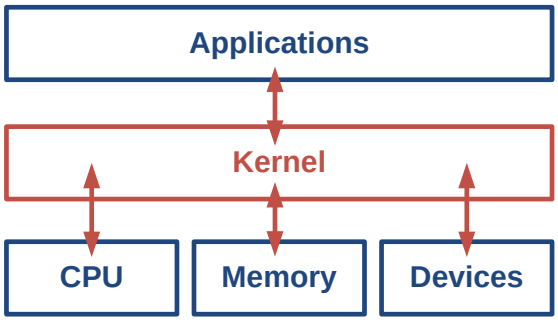


Figure 12: Operating system kernel

4.4 Real Time Preemptible Patch

When the Linux is used in real-time embedded systems, it is usually the Linux with RT-Preempt patch, which can guarantee the determinism of the system. Before the Linux RT Patch is described, it is important to define some basic properties of real-time operating systems.

4.4.1 Real-Time Operating System (RTOS)

The main difference between a classical operating system and a real-time operating system is that the real-time operating system is deterministic. For example, in the classical operating system, it is possible that the GUI sometimes freezes as the processor is busy with other operations, but this has

no fatal consequences. On the other hand, when the industrial robot moves from its operational space, it has to be immediately detected and act to prevent any damages or harms and it is not acceptable to wait for some computation to be done first. Therefore, real-time operating systems shall be used to guarantee quick responses. However, "Real Time" does not mean a fast computation, but it means quick responses with minimal delays to important events (a guarantee of worst-case times, deadlines, determinism). [7]

To achieve this, several features are introduced in the kernel of operating systems. It is for example the special scheduler for RT tasks (for example the Rate monotonic scheduler or the Earliest deadline first scheduler) which ensures meeting the deadlines and the preemptible kernel.

Real-Time operating systems can be divided into two groups according to the deadline type:

- Hard RTOS - All deadlines have to be met under all circumstances, otherwise the consequences are catastrophic, for example the controller in a plane. An interesting thing is that the analysis of the behavior of this kind of the RTOS is much more easier than in the second case.
- Soft RTOS - majority of deadlines have to be met, but some deadlines can be occasionally missed and the consequences will not be so catastrophic. But it is really tricky to define the term "occasionally" correctly. It has to be done from case to case.

4.4.2 Real-Time Preempt Patch for Linux kernel

A real-time solution for the Linux kernel is called RT-preempt patch. The advantage is that the user-space application does not care whether it runs on a pure Linux or a Linux with the RT-preempt patch installed, but the real-time behavior is better in every case. [8]

The list of features included in this patch:

- Locking primitives inside the kernel are made preemptible.
- Critical sections inside the kernel (for example a spinlock protection) are preemptible, but non-preemptible critical sections are still possible (using a raw spinlock).
- The implementation of the priority inheritance for spinlocks and semaphores inside the kernel.
- Interrupt handlers are made preemptible - each handler is now running in the preemptible kernel thread.
- Change of Timer API - introducing high-resolution kernel timers and a separate timer for timeouts.

4.5 Kernel module and device tree basics

In this chapter basic information about kernel modules and a device tree are mentioned, because this knowledge is important to understand following sections (the Section 6 in particular).

4.5.1 Kernel module

The kernel module is an extension of the kernel itself, which can be loaded during the booting time (like a driver) or on demand when the Linux is running (without a reboot). This is a great way of having the customized kernel without any great effort (for example remove unnecessary modules to reduce the size of kernel). Another advantage of the kernel modules is that the operating system's kernel does not have to be a monolithic piece of code which would lead to hardly maintainable code. New drivers and modules would have to be hardcoded to the kernel source code and therefore it would

make it too large. Thanks to modules, the kernel is just a reasonable-sized piece of code which can be extended. [13]

The code of the kernel module has to be implemented in a standardized way to ensure the portability among different versions of the kernel. All kernel modules have to include the file linux/module.h and implement initialization and exit functions, this is mandatory. These functions are called by the kernel using macros *module_init* and *module_exit* (contained in linux/init.h) when the kernel module is loaded/unloaded to/from the system. It ensures the initialization and deinitialization of necessary resources for the module.

When the kernel module is successfully loaded, some interface usually has to be implemented to communicate with the user-space application. This can be easily done by implementing file operation functions (and then fill struct file_operations which can be found in the linux/fs.h) that include for example open, close, read, write and IOCTL functions. To be able to call any file operation function, it is necessary to have a file which is assigned to the kernel module. This file can be created either by the user using the command line or by the kernel module itself, which is a preferred way (for example node in the /dev/ directory).

When we talk about the platform device driver (which means the driver of some hardware that can be found in the device tree), it has to implement some additional functions like a probe and remove functions (filled to the struct platform_driver) contained in a linux/platform_device.h altogether with a definition of a table which contains compatible strings for device tree parsing (see the paragraph below for the explanation).

4.5.2 Device tree

The device tree is a data structure which describes the devices present in the system (which includes the processor peripherals as well as the external hardware connected to the processor). By parsing this structure, the Linux kernel knows for example how much memory it has, where the modules like DMA or SPI are located, which of them shall be enabled and what driver shall be used to control them. As the name suggests, the structure is represented by a tree (in the memory it is represented by a tree of linked lists). Each node has a list of properties of several types like a string, signed/unsigned integer or array.

After a new device node is added to the device tree, the compatible string property usually has to be placed here to inform the kernel which driver should be used when the node is discovered. Other properties are filled according to device driver needs, which can be found in the documentation or directly in the code of the driver.

5 Linux installation on Profibus Analyzer hardware

As mentioned before, the RT-Linux will run on the Profibus Analyzer. The reason for using Linux for the embedded development is to make a comfort for the developer - using a stable system with the clearly defined API, portable and hardware independent code, using networking, multithreading and memory management. To take advantage of using the Linux for embedded platforms, a Processor SDK (Processor Software Development Kit) is provided to support plenty of platforms.

5.1 Processor SDK

Texas Instruments provide their own Processor SDK, which contains not just a typical BSP (Board Support Package) with a bootloader, the Linux kernel and a filesystem to be able to create the whole system from the scratch, but also pre-build images for the Sitara processor family and examples to speed up the development and start using Linux on Sitara immediately.

The processor SDK contains (see also the figure 13 below the list):

- U-boot source codes (and its configuration files).
- Linux kernel source codes (and its configuration files).
- Linaro toolchain for cross-compiling.
- Scripts for the automation, for example the booting SD Card creation and other tasks.
- Documentation.

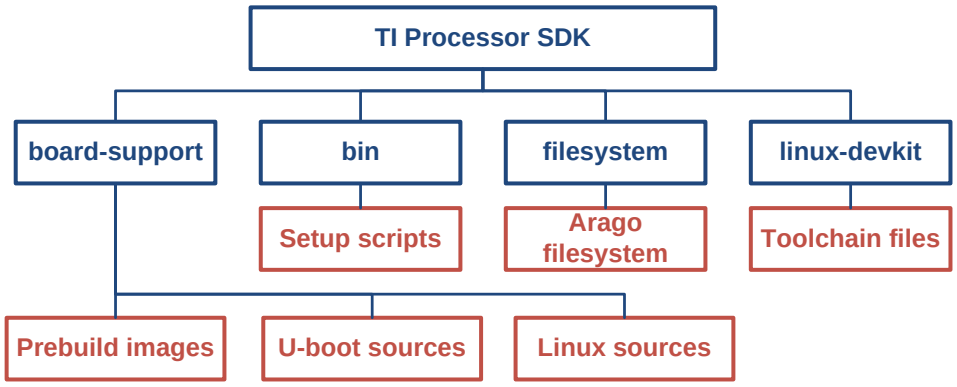


Figure 13: SDK containment overview

Although there is the support for Sitara processors, a lot of work still has to be done because of the hardware customization. The SDK is written to support TI hardware platforms (EVMs - Evaluation Modules), but if there is for example another DDR3 memory connected, it is necessary to update its timing in the U-boot or if the UART console pins are desired to be in another place (pins) it is necessary to change UART configuration.

5.2 Booting procedure

To understand the complexity of a porting procedure, it is necessary to know what exactly is happening after the device is powered up - from loading the operating system image to the memory to the peripheral initialization.

The booting of the Sitara processor includes four steps - a primary bootloader, a secondary bootloader, an U-boot and a kernel boot. Each step (except for the primary bootloader, which is hardcoded in a factory) has a potential impact when the code is ported to another platform.

5.2.1 ROM (primary) bootloader

The first phase of the booting procedure strongly depends on a type of processor. In case of the Sitara processor family, there is an option to select the boot mode by the configuration of strapping pins (called SYSBOOT [15:0]). It is possible to configure the initial crystal frequency, specify the source of an image (in our case SYSBOOT[4:0] pins are hardwired to sequence 0b10011, which gives the source boot sequence NAND (if not found continue to the next source) -> NANDI2C (not supported by hardware, certainly not found) -> MMC0 (preferred way of boot) -> UART0), enable/disable CLKOUT1 and specify options for NAND booting etc.

The Sitara ROM bootloader behavior is described in the figure 14. The code reads SYSBOOT pins, follows the sequence of source devices and tries them one by one until it reaches the end of the device

list. If the image is found in the device, then the code is loaded into the memory and the processor jumps to the initial address. If the image is not found in any device from the list, the software jumps to the infinite loop and waits for reset from the Watchdog. [10]

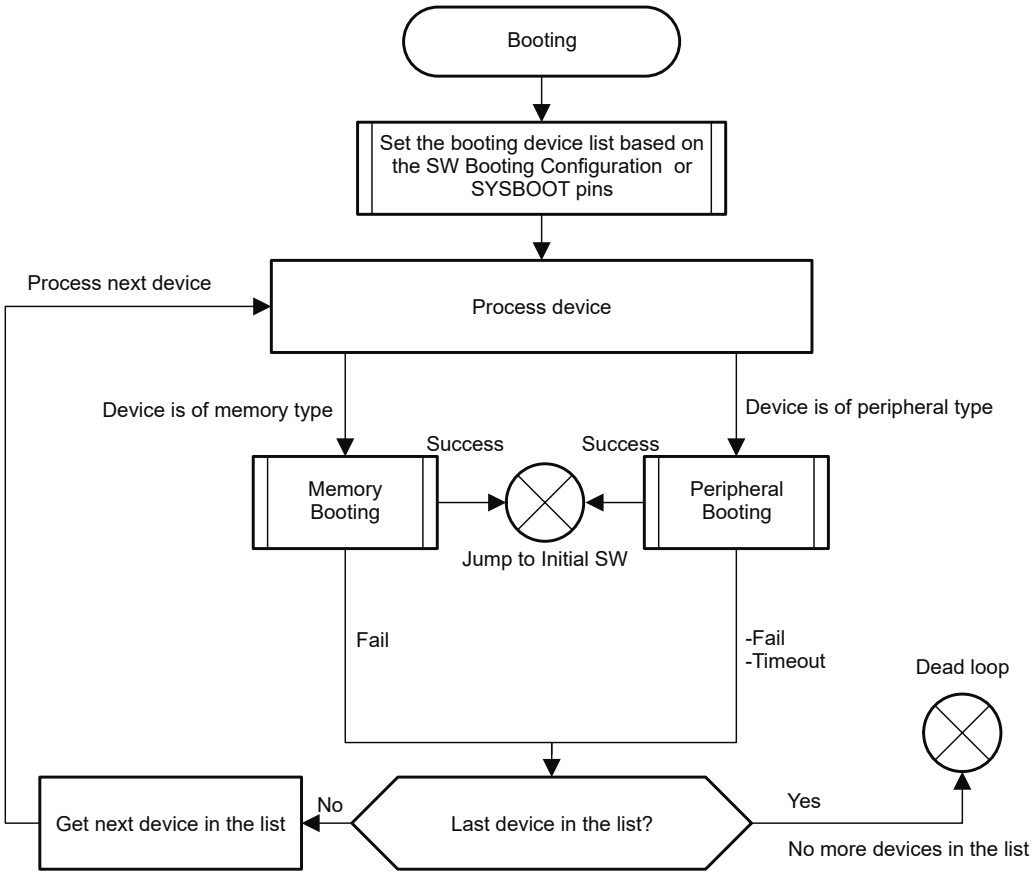


Figure 14: Sitara ROM boot code [10]

5.2.2 Secondary bootloader

When the device is found and the ROM boot code is ready to copy data to the processor RAM, we face a small problem - the size of the internal RAM is only 128kB (in fact it is a little smaller for the usage due to 18kB of the ROM code at the end of the memory and a secured first 1kB of the memory - it leaves exactly 109kB of the memory), but the U-boot code is quite complex and it does not fit (for example the prebuild U-boot image in SDK has 632 kB). The solution is to use the two-stage U-boot design which implements the primary bootloader and the final U-boot in separated binaries (therefore two stages of booting process).

The first generated binary is called MLO (minimal loader), SPL (secondary program loader) or sometimes an x-loader. The purpose is to initialize necessary peripherals (clocks, GPIO multiplexing, UART, GPMC and of course the setup of the DDR memory) to be able to copy the U-boot image to the RAM memory.

The second generated binary, which is generated at the same time as the SPL, is the U-boot image. Just note here that both binaries shared common source codes. [12]

5.2.3 U-boot

The U-boot is the image of the monolithic code which performs necessary operations to successfully run the Linux kernel. It reads and stores (and optionally decompresses) the kernel image to the memory,

it performs the checksum to verify that the loading was successful and then it jumps to the entry point (the predefined address of the image header). When the SPL is not necessary, the U-boot is responsible for the environment setup to be able to load the kernel image to RAM. This setup includes the clock configuration, performing necessary pin mux settings (for example for the MMC) and timing configuration for the RAM (for example DDR3).

The U-boot contains its own command line, where various things can be set up. It is for instance the change of the name of the desired Linux kernel image (there can be more images in the source device memory), addresses where to load kernel image and the device tree, settings of output debug console and many other things. Some of these environment variables such as addresses are passed to the Linux kernel. [12]

5.2.4 Kernel

When the Linux kernel starts, it performs the low-level system setup first. It initializes the virtual memory, installs interrupt handlers, starts the scheduler, decodes the device tree, loads kernel modules and finally sets up the user space context.

5.3 U-boot porting

List of actions which are usually needed in case of U-boot porting:

- Set Build Configuration - by the modification of build configuration the interface used by U-boot is selected (for example GPMC, DDR) and the SPL usage is enabled or disabled.
- Set Pin Multiplexing - pin multiplexing is related to the previous point, because the interface used by U-boot needs to have GPIOs configured in a proper way.
- Set Platform Data - it contains information related to the custom hardware and which cannot be found automatically (for example DDR3 timing).
- Change Board level code - this code performs necessary initialization of peripherals and interfaces using the Platform Data and it is used as the Board Adaptational Layer.

In this case, changes in U-Boot code were made because of the different DDR3 memory chip and because no card detection signal from the SD-Card is connected to the Sitara processor. Another minor changes were made in configuration files.

5.3.1 Configuration

Most parts of the configuration for Profibus Analyzer were reused from the AM335x Evaluation Module. The difference is the specified Device tree which shall be used during the kernel boot. CONFIG_SPL_BUILD is defined to build both the SPL and the U-boot binaries.

The memory location for the FDT (Flattened Device Tree) was changed and the memory range was updated in arguments which are passed to the kernel from the U-boot because of the different size of the DDR3.

5.3.2 Board init

Board specific source files were completely rewritten according to source files for the Beagle Bone Black. Source codes contain DDR3 settings, a configuration of initial LED state to signalize that the U-boot runs on the Sitara processor and a configuration of the Watchdog module to reset the Sitara in case of

problem (in case of some initialization problem the program usually ends in a dead loop and therefore the Watchdog is really useful).

Pin multiplexing was also configured in the Board specific source files.

5.3.3 Card Detect Signal

The special case for the Profibus Analyzer hardware is that the Card Detect signal is not connected from the SD-Card to the MMC module. Because of this, there was no other option than to rewrite few lines in the MMC driver to avoid checking this signal.

5.4 Linux kernel porting

List of actions which are usually needed in case of Linux kernel porting:

- Set Build Configuration - by the modification of the build configuration the content of kernel is specified (for example which drivers will be available, debug modes, etc.).
- Update Device Tree - by the modification of the device tree the developer tells the kernel what hardware is present and what driver shall be used.

In this case, changes in kernel source codes were made, as expected, in the device tree. Another part is, again, missing SD-Card Card Detect signal and configuration files.

5.4.1 Configuration

The kernel build configuration was completely rewritten to make the kernel as small as possible, which means that only drivers which are really needed were kept in the configuration. The debug features were used only during the development and for the final version they were disabled.

5.4.2 Device tree

The device tree was updated to support only peripherals which are necessary for the Profibus Analyzer functionality. List of supported peripherals, which are contained in the customized device tree (this tree includes am33xx.dtsi, which adds necessary features of the AM33xx processor family like the EDMA, CPU, etc. and therefore are not included in customized device tree):

- UART for serial debug console,
- MMC for the SD card,
- SPI0 for the FPGA firmware upload,
- GPMC for the memory (FPGA) access,
- Ethernet subsystem (CPSW, MAC, MDIO),
- Linux heartbeat LED,
- DDR3 memory size definition.

The MMC node was configured so the driver knows, that there is no Card Detect signal and therefore it is not a problem anymore. The memory range was updated as well.

Profibus Analyzer channel nodes were also added to the customized device tree. This is described in a 6.2.4 subsection.

6 Profibus Analyzer driver

This chapter describes the crucial part of the thesis. This chapter goes in more detail to give the reader the whole picture of the driver functionality altogether with the usage of the driver from the user-space application.

The main idea is to have a driver as a kernel module running in the background to prepare the data for the user-space application which will forward the data to the PC application via the Ethernet interface. Driver will automatically read the data from the FPGA and transfer them to the DDR3 memory. After receiving some amount of data, the driver will inform the user-space application that the data are ready to be sent. The key feature is to make this part of the transfer stack as fast as possible.

6.1 Driver design

According to the device architecture, it was intended to implement driver based on a producer-consumer scheme. In this subsection, the design of the driver together with the interface which will be used from the user-space application is described. Also, to describe the design completely, the way of loading the driver to the kernel is introduced.

6.1.1 Initialization/deinitialization

The description of the driver design should begin with the information how and when is the kernel module and the platform driver respectively inserted into the kernel.

First important thing is that the driver has to be installed in the booting memory. If the driver is successfully installed, the compatibility property string table is registered during the boot time altogether with the device number and class.

Whenever during the device tree parsing (during boot time) the compatibility string property is reached, the kernel finds the compatible driver by matching it with the compatibility string specified by the driver which was initialized before. Then the appropriate driver's probe function is invoked, the driver takes control and the device is initialized. The node in the /dev/ directory is usually created in this time.

This situation is captured in the figure below and it is divided into two parts:

- Module init/exit - the part where the driver is loaded to the kernel. The class is created, the device number is registered and the compatibility string property is passed to the kernel.
- Channel init/exit - the part where the device is found in the device tree and the device platform driver is chosen by the compatibility property. The driver then parses the device tree node and stores its data to the structure to be used later when the device is used. At the same time some of the resources are reserved, for example the memory is allocated for the cyclic DMA buffer. Each Profibus Analyzer channel has its own device tree node. This part is called separately for each node and therefore an entry in the /dev/ directory is created for every single channel.

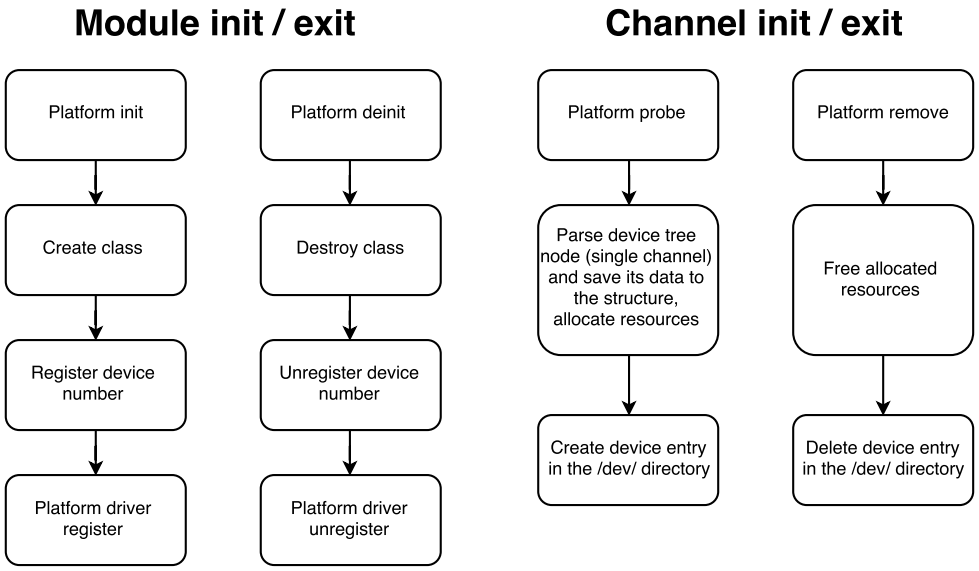


Figure 15: Loading driver to the kernel

6.1.2 Control interface

As mentioned before, each Profibus Analyzer channel has its own device tree node and therefore the channel init function is invoked multiple times, which means that there are different entries in the /dev/ directory for each channel. Therefore, the interface has to be designed to be used for each channel separately without affecting the others.

The preferred way of communication with the single channel is to use standard file operations, in our case, following functions are implemented:

- **Open** function is intended to initialize the single channel of Profibus Analyzer to be fully prepared to start capturing the data from the FPGA. The DMA channel is opened (but the transfer initialization is performed after the channel start function is called), the GPIO is exported and the appropriate interrupt is requested and the internal FIFO, which is used to hold the queue of the buffer information to be sent to the user space, is initialized.
- **Release** function is used to free the exported GPIO together with the requested interrupt. The internal FIFO is cleared, the Watchdog timer is deleted and the DMA channel is released.

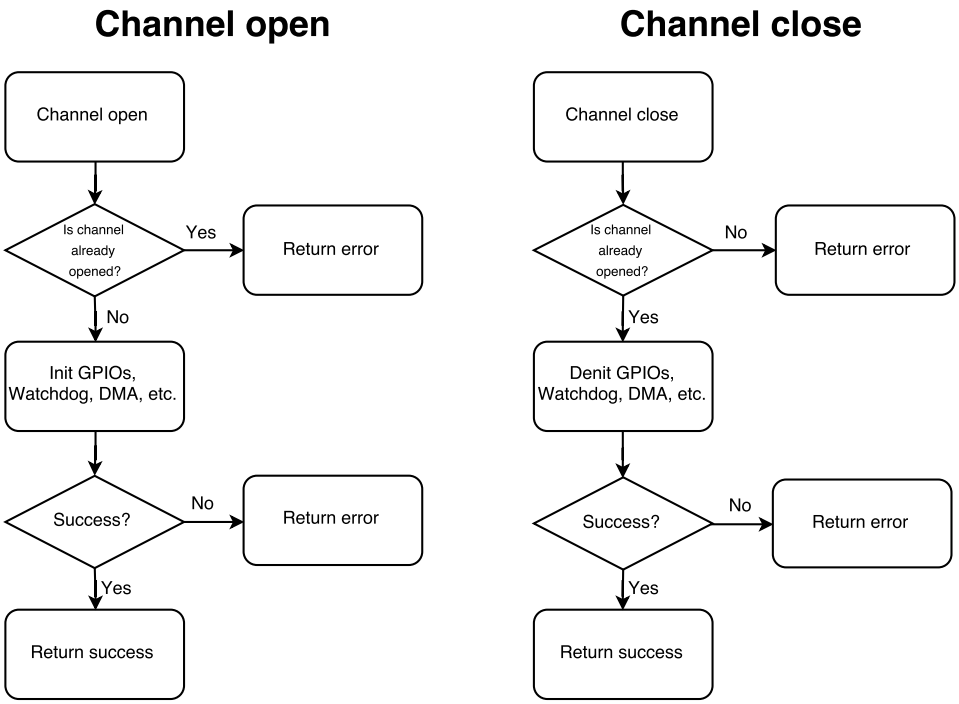


Figure 16: Open close functionality

- **Read** function is designed to be as fast as possible. Therefore, just the information about the buffer (address and size packed in the structure) is passed from the driver to the user-space application. By using the shared memory concept between the driver and the application (using the mmap functionality), the user-space application can access the data from the driver. This approach was chosen to avoid the unnecessary memory copy from one buffer to another and therefore to avoid the CPU overload. The first structure in the internal FIFO is returned to the user. It is necessary to keep the order of used buffers. It prevents the situation that the buffer which was not read yet by the application is overwritten by the DMA.

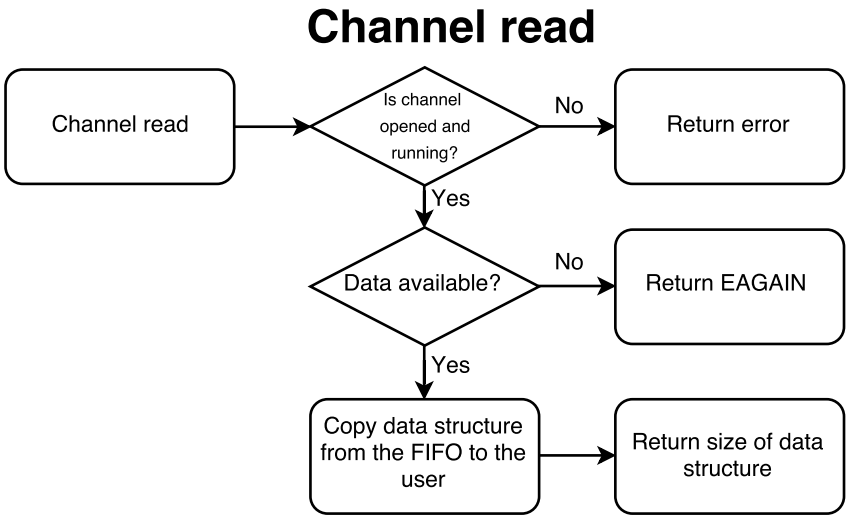


Figure 17: Read functionality

- **Unlocked_ioctl** (Input Output Control) interface is used to send/receive commands to/from the driver. The following commands are used:
  - Start is used to start capturing data from the Profibus channel.
  - Stop is used to stop capturing data from the Profibus channel.
  - Set configuration is used to configure FPGA settings such as Profibus channel speed.
  - Get configuration is used to read current settings from the FPGA.
  - Get buffer map is used to get buffer map, which was initialized during the device probing. The main idea is to pass buffer locations to the user-space application so it can be statically mapped before the channel is started. This approach saves a lot of time, because the mapping is time-consuming, and therefore just a single mapping at the beginning saves time and increases data throughput.
  - Get available bytes is used to return the number of available bytes in the current buffer.
  - Set the trace level (choose from debug/warning/error trace level).

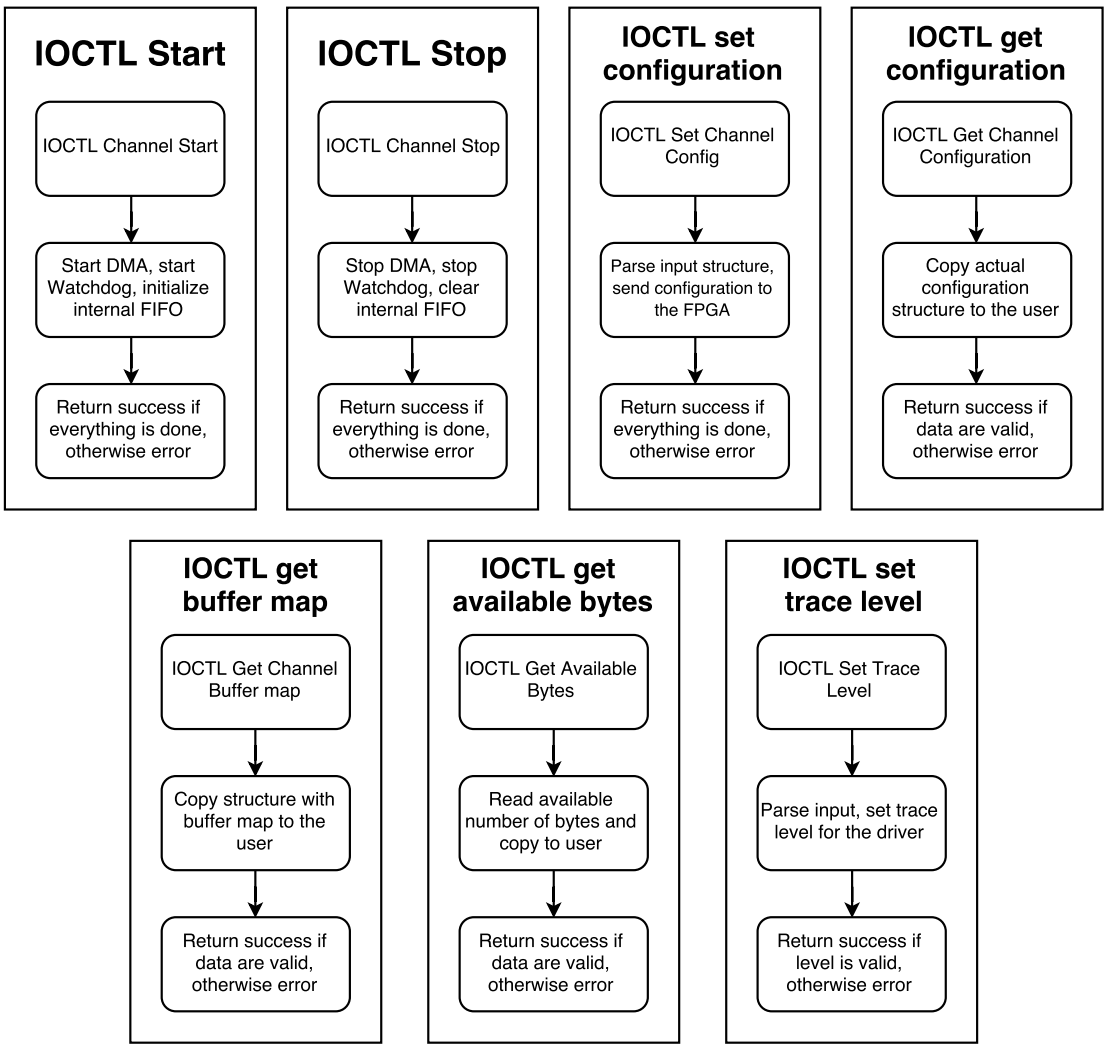


Figure 18: IOCTL commands

- **Poll** is implemented to support the *poll* and *select* functions, which can be used from the user-space application to check whether the data are available.

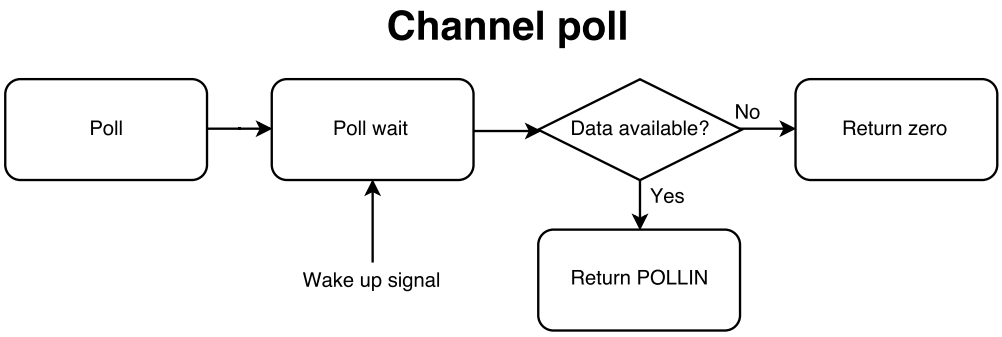


Figure 19: Poll functionality

6.1.3 Functionality

When the channel is configured and it is running, data should be transferred to the CPU memory automatically without any software support by using the DMA event trigger which is controlled by the FPGA chip. This is done by the appropriate configuration of the DMA controller, which will then transfer 8 bits on every trigger from the FPGA (the trigger comes as long as there are some data available in the FPGA's FIFO).

The DMA is also configured to cause interrupt after the buffer is full (DMA automatically switches buffers) and the handler is then responsible to signalize that there are some data available in the internal FIFO ready to transfer to the poll function (and through driver to the user-space application which is using the *poll*/*select* functions).

To avoid the situation when the data rate is really slow (for example some misbehavior on the bus) and the DMA interrupt occurs just occasionally which would then slow the transfer of the data to the PC application, the Watchdog timer was introduced. After some predefined time, the Watchdog interrupt occurs and buffers are checked for available data. If some data are available, they are placed to the internal FIFO, the signal is sent to the *poll* function and then the application has to take care about sending the data to the PC. The DMA callback always resets Watchdog counter to avoid unnecessary interference of the standard behavior.

The last supported functionality is handling the external trigger. This is done by installing the GPIO interrupt which is then responsible for starting the channel.

Just note here that described functionality is separated for every channel and therefore there are no shared resources among channels.

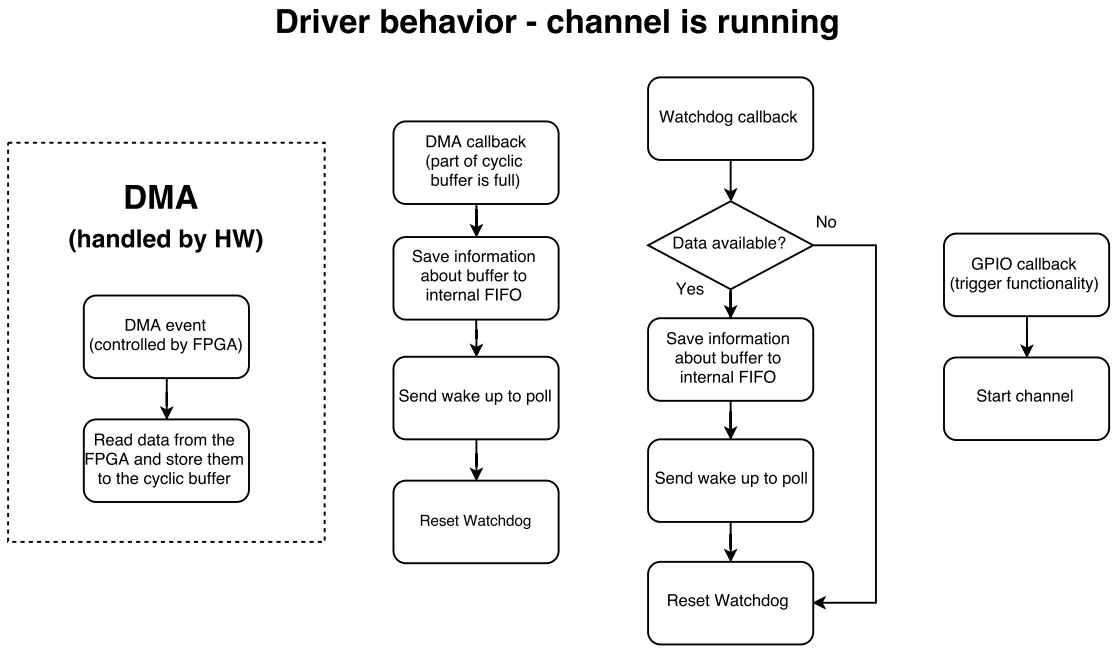


Figure 20: Diagram of driver behavior

6.2 Driver implementation

The implementation of the core features is described in this subsection. It is not worth to describe every detail, so the focus will be on the file structure, the interface to the user space and the DMA functionality, which unexpectedly required some additional features to be added.

6.2.1 Source code structure

It was decided to divide the functionality into more source files due to better readability, extensibility and maintenance of the source code. The source code is created by following files:

- C files
  - **profibus_analyzer_driver.c** contains the mandatory functionality for the platform device driver like a file operations structure, platform driver operations and module init/exit functions.
  - **profibus_analyzer_fcns.c** contains the whole functionality used for the data flow.
- Headers
  - **profibus_analyzer_driver.h** contains the definition of the Profibus Analyzer structure which holds information about every single channel.
  - **profibus_analyzer_fcns.h** contains only prototypes of the functions in complementary .c file.
  - **profibus_analyzer_ioctl.h** contains the interface which is used to communicate with the user-space application.

6.2.2 Data transfer functionality

The most challenging part of the Profibus Analyzer driver implementation was to implement the data flow strategy which was described above (in the 6.1.3). The situation of the continuous data stream from the FPGA is captured in a figure 21.

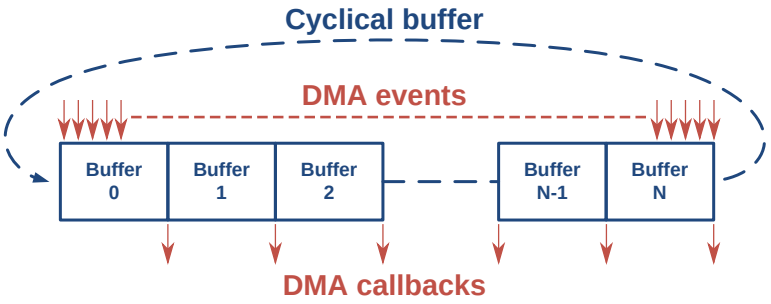


Figure 21: Cyclical buffer with DMA callbacks

The DMA callback always appears before the Watchdog overflow and so only the buffer ID is necessary to determine the buffer correctly. As usual, nothing is perfect and therefore the driver has to deal with the situation of a non-continuous data stream from the FPGA. This situation is captured in the figure 22.

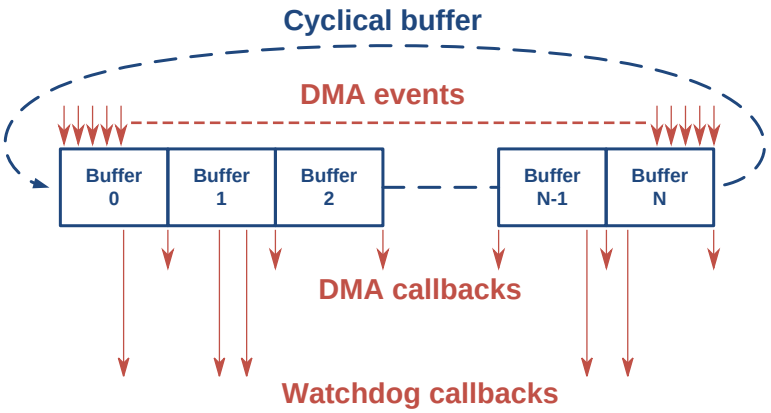


Figure 22: Cyclical buffer with DMA and WDG callbacks

The buffer is filled by the DMA, but the data rate is too slow. After some time, the Watchdog timer overflows and therefore the Watchdog callback is performed. Now, the driver has to find how many bytes have been transferred by the DMA transfer from the FPGA to the buffer (unfortunately, the DMA API does not provide such functionality and therefore it had to be implemented in a different way, which is described in the following section), it has to compute the buffer address and then put it in the internal FIFO.

As can be seen from the figure 22, because of the Watchdog callback, the buffer is no more aligned and callbacks which come after it have to deal with this situation by reading the previous source of data and then computing the buffer offset and size.

Number of transferred bytes

During the implementation of the driver, the function to read the number of transferred bytes had to be implemented because there is no support for this kind of action by the DMA driver. To implement

this, the EDMA (Enhanced Direct Memory Access) controller area, which is present in the Sitara, had to be accessed directly by the driver.

The EDMA is a RAM-based architecture module, which in this case means, that the EDMA module contains a PaRAM (Parametrization RAM), where transfer parameters are stored. The PaRAM structure is captured in figure 23.

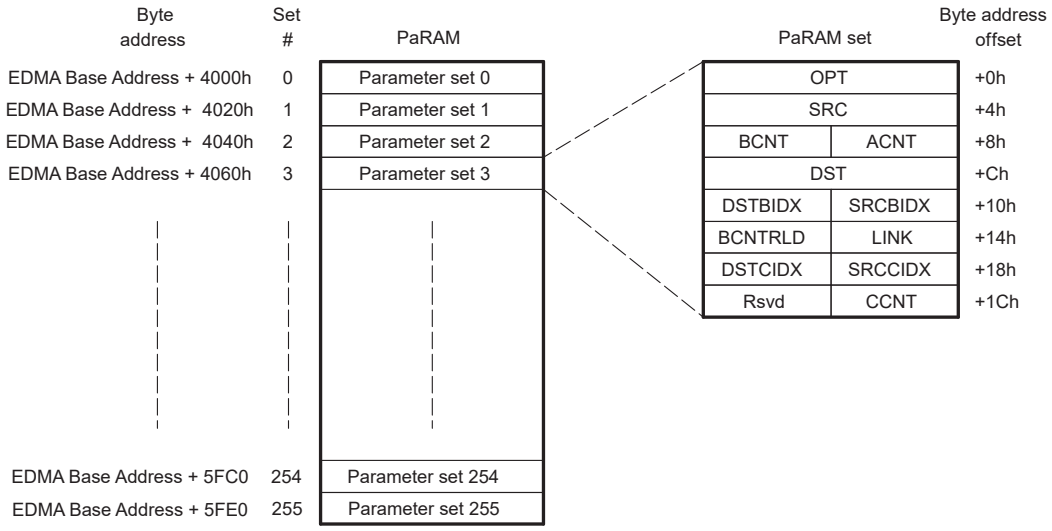


Figure 23: EDMA PaRAM[10]

As can be seen at figure 23, there are 256 sets in the RAM. Each PaRAM set is 32B and contains information needed by the single transfer. The most important is the BCNT field, where the remaining number of words can be found (for different configuration the CCNT and ACNT play important role as well). The cyclic DMA buffer is implemented as a cyclic chain of PaRAM sets (where last points to the first) and the currently active set is always copied in front of this chain.

Finally, the implementation was not so difficult. The PaRAM was mapped using the ioremap functionality, so the driver can access it. The copy of the PaRAM set is found during the channel start and the pointer to this PaRAM set is stored. The actual number of transferred bytes can be always determined by reading value from the PaRAM set (on a static address), because the actual set is always copied to this place. [10]

6.2.3 Control interface to the user-space

Almost everything which is needed by the user-space application is contained in the IOCTL header. The header has to be the same for both, the driver and the user-space application.

Channel open/close

The channel open/close description can be found above. After the channel is opened, the descriptor is returned and then it can be passed to all the control functions like channel read and so on.

Buffer map

The buffer map structure is returned after the IOCTL command *IOCTL_GET_BUFFER_MAP* is called. The purpose is to allow the user-application to map the buffers from the physical address to the

virtual address and therefore to allow the user-space application to access the driver buffers, because the mmap function takes some time and doing it after every read would not be comfortable and it would be slow.

The IOCTL command returns the structure (in the frame below), which contains following fields:

- *base_addr* contains the physical address of the base of the cyclical buffer.
- *single_buf_size* holds the size of the single buffer.
- *number_of_bufs* holds the number of smaller single buffers in the cyclical buffer. Therefore the size of the cyclical buffer is the *single_buf_size* multiplied by the *number_of_bufs*.

```
typedef struct {
      u32 base_addr;
      ssize_t single_buf_size;
      unsigned int number_of_bufs;
} p_a_buffer_map_t;
```

Figure 24: Structure which is returned from buffer map function

Channel read

The read function returns the structure (in the frame below), which contains following fields:

- *buffer_id* contains the ID number of the buffer.
- *base_addr* contains the physical address of the buffer. It is useful only when the buffer is not aligned, which happens only if the Watchdog interrupt occurred before.
- *size* holds the valid data size in the buffer (maximal value is the size of a single buffer).
- *align* is enum type which supports two values - *e_data_aligned* and *e_data_misaligned*. The purpose of this field is to simply identify, whether the *buffer_id* or the *base_address* shall be used to locate the buffer.
- *source* is enum type which supports two values - *e_data_source_wdg* and *e_data_source_dma*. The original usage was only for the debug, but at the end, it can signalize to the application that something is wrong, because the incoming data stream from the FPGA was interrupted.

```
typedef struct {
      unsigned int buffer_id;
      u32 base_addr;
      ssize_t size;
      t_p_a_data_alignment align;
      t_p_a_data_source source;
} p_a_read_data_t;
```

Figure 25: Structure which is returned from the read function

Channel configuration

The configuration is set or get after the IOCTL command *IOCTL_CHAN_SET_CONFIGURATION* or *IOCTL_CHAN_GET_CONFIGURATION* is called. The purpose is to allow the user-application to configure the FPGA behavior.

The IOCTL command requires/returns the structure (in the frame below), which contains following fields:

- *speed* contains the speed of the Profibus Channel.

```
typedef struct {
     int speed;
} p_a_config_t;
```

Figure 26: Structure holding the configuration of PFGA

Channel input/output control

The rest of IOCTL commands are:

- *IOCTL_CHAN_START/IOCTL_CHAN_STOP* command starts/stops the channel.
- *IOCTL_GET_AVAILABLE_BYTES* returns the number of available bytes in the current single buffer.
- *IOCTL_SET_TRACE_LEVEL* configures the trace level (possible levels are defined by enum as *p_a_trace_level_name*, where *name* can be debug, warning, error or none).

6.2.4 Device tree

For every additional hardware there has to be a support in the device tree to make it work correctly. For the Profibus Analyzer, the GPMC and pin multiplexing settings have to be changed and Profibus Analyzer Channel nodes have to be added. These changes are described in this subsection.

General Purpose Memory Controller

GPMC settings are an important part of the driver, because the memory access to the FPGA could not work if the GPMC is not configured in a proper way.

Important settings for the Profibus Analyzer driver which have to be present in the GPMC node:

- There are about 6 registers which are responsible for timing settings for every single bank of the GPMC. These values have to be configured in a proper way, because the timing is hardwired in the FPGA.
- The address offset for the FPGA bank and the chip select number.
- Bus width and address/data multiplexing.
- Used GPIOs.

The whole GPMC node can be found in the Appendix, GPMC settings.

Pin multiplexing node

The pin multiplexing node can be found in the pin multiplexing settings part of the device tree. The appropriate pin has to be configured for the correct mode and direction to work as a trigger event for the EDMA. The example below is the event pin for the Profibus Channel DP 1 (the DP2 and PA pins are filled accordingly).

```
xdma_event_pin_dp_1:  xdma_event_pin_dp_1 {
      pinctrl-single,pins = <
            /* xdma_event_intr_0 */
            AM33XX_IOPAD(0x9B0, PIN_INPUT | MUX_MODE0)
      >;
};
```

Figure 27: Pinmux node in the device tree

Profibus Analyzer node

The Profibus Analyzer node in the device tree describes used hardware resources (like DMA channel) and FPGA settings. An example of this node can be found in the frame below. The example below is the Profibus Analyzer node for the Profibus Channel DP 1 (the DP2 and PA channels are filled accordingly).

```
profibus_analyzer_channel_dp_1 {
      pinctrl-names = "default";
      pinctrl-0 = <&xdma_event_pin_dp_1>;

      #address-cells = <1>;
      #size-cells = <1>;

      compatible = "profibus_analyzer_channel";
      pa-chan-name = "p_a_chan_dp_1";

      pa-single-buf-size = <0x1000>;

      dmas = <&edma_xbar 35 0 28>;
      dma-names = "rx";

      pa-fpga-gpmc-base = <0x10000000>;
      pa-fpga-data-reg = <0x20000>;
      pa-fpga-count-reg = <0x20004>;
      pa-fpga-config-reg = <0x20008>;

      pa-chan-trig-pin = <112>;
};
```

Figure 28: Profibus Analyzer Channel node in the device tree

More detailed description of single properties in the node follows:

- *pinctrl-names* and *pinctrl-0*: These properties are linked to the pin multiplexing node and they provide the correct configuration of the EDMA trigger pin.

- *compatible*: as mentioned before, the compatible string property defines which driver shall be used by the Linux kernel, when the node is reached during booting time.
- *pa-chan-name*: This string property defines the device name which will appear in the /dev/ directory.
- *pa-single-buf-size*: It defines the single buffer size (the cyclic buffer is created by 16 smaller buffers, which has the single buffer size).
- *dmas*: It is important for the driver to use the appropriate EDMA channel, which is triggered by the event pin. This property is in <&edma_xbar open_channel_number 0 crossbar_event_number> format (these numbers can be found in the Sitara datasheet, section 11.3.19 EDMA Events).
- *dma-names*: The dmas array entry has to be named and therefore the dma-names is filled by the "rx" string. The driver parses this string as well, so there can be filled almost everything and the driver will deal with it.
- *pa-fpga-gpmc-base*: This property is connected with the GPMC node settings. This value has to be exactly the same as the base address of FPGA in the GPMC node.
- *pa-fpga-data-reg*: It contains the address of the data register (FIFO) in the FPGA.
- *pa-fpga-count-reg*: It contains the address of the data counter register in the FPGA.
- *pa-fpga-config-reg*: It contains the address of the configuration register in the FPGA.
- *pa-chan-trig-pin*: It holds the GPIO number of pin which is responsible for triggering the start of capturing the data (pin number is computed as 32x + y for GPIOx, pin y).

6.3 Example of driver usage

To show the usage of the designed driver, in this section is the simplest possible example. The frame on the left side contains opening and closing functionality altogether with the IOCTL example how to get the buffer map and how to start the channel. The frame on the right side illustrates how to correctly use the *select* function, which returns every time when some of file descriptors from the set are ready to provide the data. In this step, all channels can be combined together with any other sockets.

```
p_a_buffer_map_t buf_map;
int rv, fd;

fd = open("/dev/p_a_chan_dp_1", O_RDWR);
if (fd < 0) {
      /* Handle error */
}

rv = ioctl(fd, IOCTL_GET_BUFFER_MAP,
&buf_map);
if (rv < 0) {
      /* Handle error */
}

/* Here we have buffer descriptor and we
can do mmap to virtual space */

rv = ioctl(fd, IOCTL_CHAN_START);
if (rv < 0) {
      /* Handle error */
}

/* Here shall be the main part of the
application.  See the right frame for the
reading sequence */

rv = close(fd);
if (rv < 0) {
      /* Handle error */
}
```

```
fd_set rset;
p_a_read_data_t read_data;

/* Ususualt this is placed in a loop */
FD_ZERO(&rset);
/* Add descriptors to the set */
FD_SET(fd, &rset);

/* Wait for descriptors to become ready
*/
rv = select(fd + 1, &rset, NULL, NULL,
NULL);
if (rv < 0) {
      /* Handle error */
}

/* Check which descriptor is ready */
if (FD_ISSET(fd, &rset))
{
      /* This one is ready */
      rv = read(fd, &read_data,
            sizeof(p_a_read_data_t));
      if (rv < 0) {
            /* Handle error */
      }
}

/* Here we have information about buffer,
size, source of data and the data
alignment */
```

6.4 Validation of driver implementation

The verification process included the special FPGA bitstream and the user-space application.

6.4.1 FPGA test design

For the verification procedure, the special FPGA bitstream was developed. It includes the same functionalities as the regular bitstream with a single change - it does not capture Profibus telegrams, but it generates the incrementing number sequence instead. In this way, it was really easy to determine whether some data were missing, whether some buffers were filled in a bad way or other possible errors.

6.4.2 User-space test application design

The user-space testing application included the test of basics functionalities such as opening the channel when it is already opened, sending stop command when the channel is not running and another wrong combinations, which has to be handled in a correct way by the driver.

Another test was the test of the data flow. The test bitstream was loaded to the FPGA and a channel was started. The application read the data and checked whether the sequence is correct and whether the data size is correct. It also checked the buffer position and printed results to the console.

7 Profibus Analyzer Application

The Profibus Analyzer Application part describes the Linux user-space application, which is the last firmware part of the transfer chain running on the Sitara processor. The application uses the Profibus Analyzer Driver designed and implemented before in this thesis. The functionality of the driver is described in previous chapter.

7.1 Application design

The main function of the Profibus Analyzer Application is forwarding the data from the FPGA (more precisely from the Profibus Analyzer Driver) to the control PC running the control application. Another function is to allow controlling of the Profibus Analyzer from the PC application (which includes turning the specific channel on/off, selection of baudrate, etc.) and getting information (for example the FPGA bitstream version information) and diagnostics (for example last error or debug logs) from the Profibus Analyzer.

The description below is divided into two parts - the initialization part, where the necessary actions to initialize the FPGA and server functionality are described and the server/control loop part, where the main control loop (which behaves like a server) is described.

7.1.1 Initialization

The initialization part begins with the bitstream loading to the FPGA. This is the first and also a crucial part of the initialization, because without the successfully loaded FPGA there is no chance to continue. The way how the bitstream is loaded to the FPGA is described below in the Section 7.3. If the loading fails, the application terminates and saves error log to the memory.

The second part of the initialization is the network connection configuration, which includes the IP address assignment to the Profibus Analyzer (to the Sitara processor) and the socket opening. The IP address, netmask and port values are loaded either from the configuration file (if it is found) or from the hardcoded values (if the configuration file is not found or if the values in the configuration file are not valid). The stream (server TCP) socket, which is used to transfer control frames (server requests and responses), is opened. After that, another three (under special circumstances there can be less than three) datagram sockets are opened. They are used to transfer Profibus telegrams data from the FPGA - one for every Profibus Analyzer channel. The opening of one socket happens only if the Profibus Analyzer channel is successfully opened, therefore this initialization step is closely connected to Profibus Analyzer channels initialization step.

The last part of the initialization is the opening and initialization of all Profibus Analyzer channels for which the Profibus Analyzer Driver is used. If the initialization of specific Profibus Analyzer channel fails, it is disabled to be used in next steps of the application, but the application can continue to run and work just with successfully opened channels.

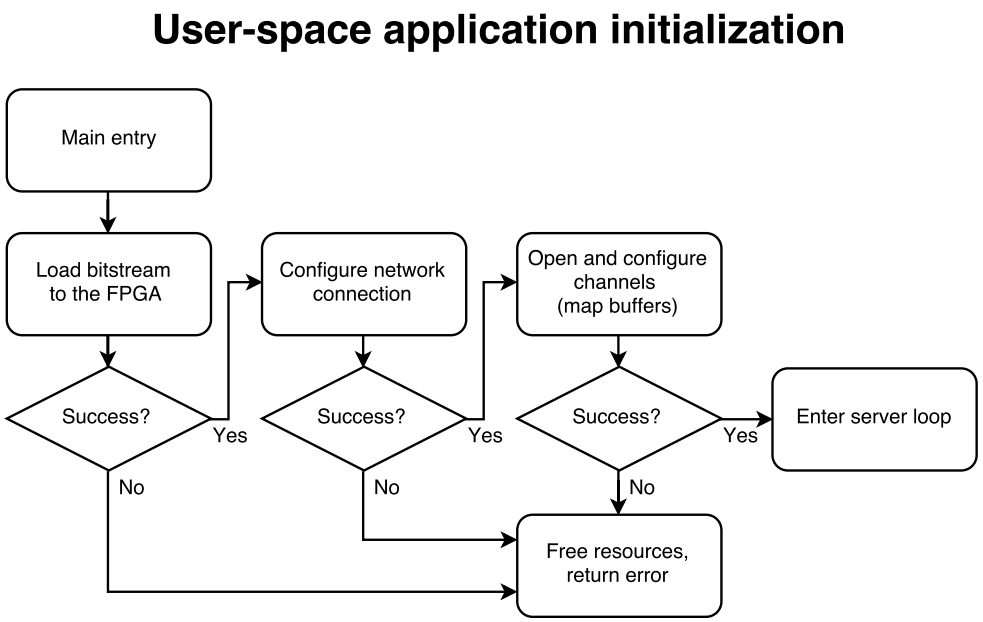


Figure 29: Software diagram for initialization part

7.1.2 Server/control loop

The server/control loop begins with waiting for the connection of a client. This part is also used if the client disconnects and there is no other client active. When the client is successfully connected, the server/control loop is reached.

At the beginning of the loop, all client sockets and all opened channels are checked whether there is an incoming request from the client or whether there are any available data which are ready to be sent to the client. If the client socket sends data to the application, it is parsed and if data contains the valid command, the appropriate action is performed and the response with the results of action is sent back to the client. If there are any data available on any Profibus Analyzer channel, they are immediately sent via the datagram socket, which is assigned to this channel.

The server also supports changing of IP address, netmask and used port, which is requested by the PC application. After this request is received, the configuration is performed and stored to the configuration file and the server is restarted.

Another supported feature is replacing the client. If the new connection request comes on the original socket, the old client is disconnected and replaced by a new one.

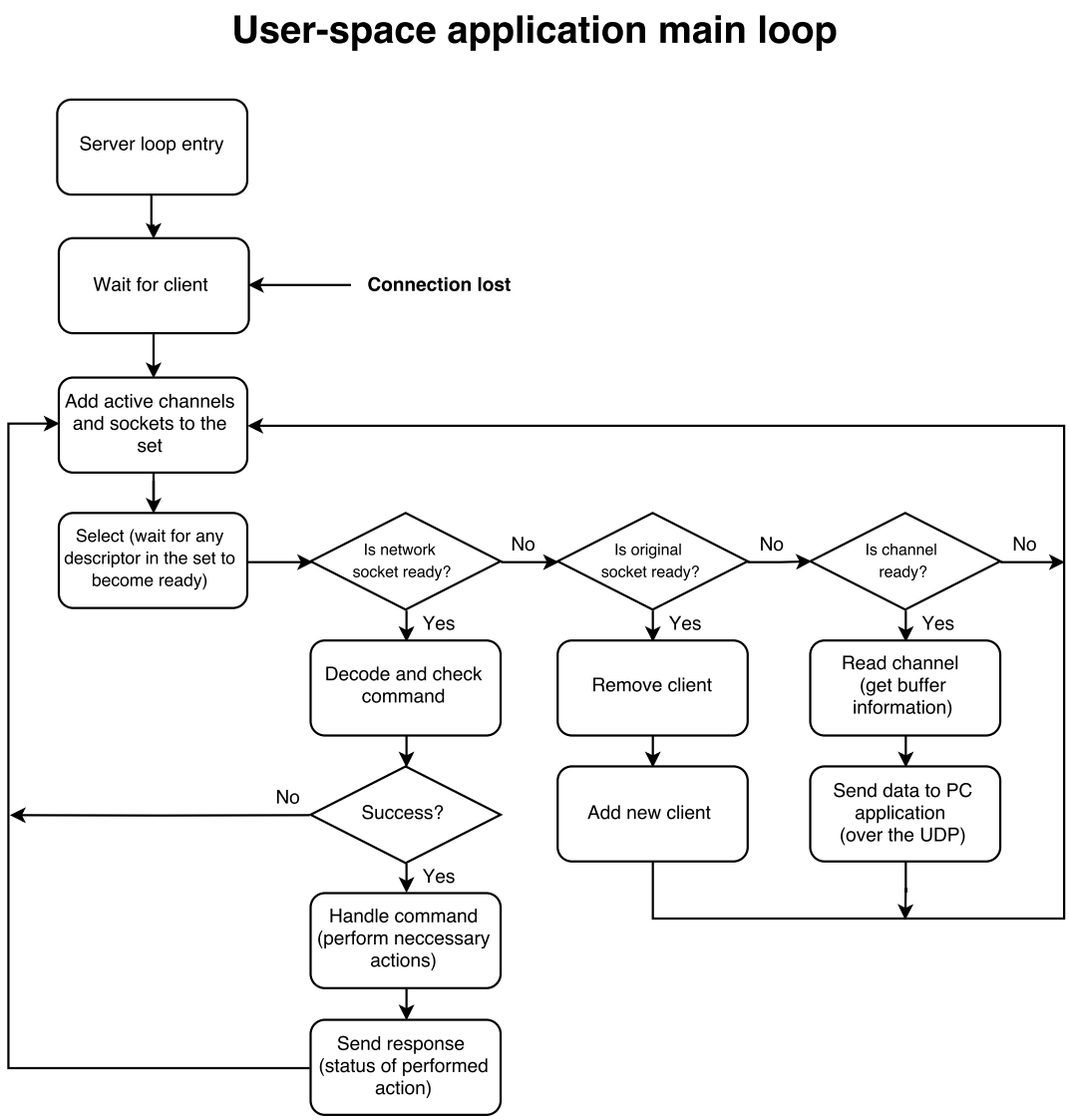


Figure 30: Software diagram for main loop part

7.2 Application implementation

The implementation section is divided into several parts, which include the description of the code structure, application configuration, program behavior and the communication protocol description.

7.2.1 Code structure

The application source code is created by following files:

- **p_a_main.c** contains initialization calls and the main server loop.
- **p_a_fpga.c/p_a_fpga.h** contain the FPGA loading code.
- **p_a_gpio.c/p_a_gpio.h** contain the GPIO driver which is used for FPGA loading and for LED control.

- **p_a_channel.c/p_a_channel.h/p_a_ioctl.h** is used to control the Profibus Analyzer Driver.
- **p_a_config_parser.c/p_a_config_parser.h** contain the configuration file loading, parsing and saving feature.
- **p_a_server_config.c/p_a_server_config.h** is used to control the network interface and sockets.
- **p_a_server_fcns.c/p_a_server_fcns.h** contain server functionalities like an incoming command decoding and sending responses and Profibus Analyzer channel data sending.
- **p_a_update.c/p_a_update.h** contain the protocol which is used to update the kernel module (Profibus Analyzer Driver) or the Profibus Analyzer application.
- **p_a_trace.c/p_a_trace.h** is used to create a trace file and insert trace messages into it.
- **p_a_common.h** contains structures which are shared among the source codes and version defines.

7.2.2 Network configuration

The network connection configuration is implemented in **p_a_server_config** and **p_a_config_parser**. When the application starts, it will try to find a configuration file (with a fixed name **p_a_config_file**). If the file is found and if it is in an expected form (example in figure 31), the IP address, netmask and port is configured according to values in the file. Otherwise, if the file is not found, default values are used. Default values are:

- IP Address 192.168.0.1,
- Netmask 255.255.0.0,
- Port 60010.

```
#Config file for the Profibus Analyzer application
ip=192.168.0.1
mask=255.255.0.0
port=60010
```

Figure 31: Profibus Analyzer configuration file

These values can be changed during the runtime by the PC control application. When the configuration is done, the configuration file is overwritten with new values.

The networking is based on socket operations. For the client connection the TCP stream is used and for the Profibus Analyzer channels data the UPD is used. The data from Profibus Analyzer channels are transferred on the UDP port + channel number, which means ports 60001, 60002 and 60003.

7.2.3 Server behavior

The main loop is based on the *select* function, which gets descriptor set (*fd_set*) as an argument and returns when some descriptor is ready. At the beginning of the loop, all running channels and all clients (it is possible to allow more connections) are added to the fd_set and then, after the *select* returns, the channel/client which is ready is found. In case of incoming data from the client, the command is decoded in the p_a_server_fcns.c file and the appropriate action is taken. When data are ready, the appropriate UDP port is chosen and then data are sent to the PC.

7.2.4 Traces

There has to be some tool for diagnostics to keep information about the state, actions which were taken, runtime problems and so on. Therefore, the trace feature was added to the source code. There are no direct prints to the console in the code - everything goes via the trace feature. Traces can be sent to the PC control application.

Traces are divided into several levels, which can be enabled/disabled separately:

- Debug (contains the information about sent data, buffers locations, etc.)
- Info (contains the information about channel opening/closing, GPIO states, configuration, etc.)
- Warning (contains the information that something goes wrong, but the application can handle it, like situations when a client is disconnected, the configuration file is not found, etc.)
- Error (contains the general information that something goes wrong, for example that the Profibus Analyzer channel cannot be opened, etc.)
- Error detail (contains the specification of the last error, for example that the Profibus Analyzer channel cannot be opened because it is already opened, etc.)

```
*******************************
*** Profibus Analyzer Started ***
*******************************
...
INFO: Configuration file opened, parsing content!
INFO: Property ip found!  Parsing value!
INFO: Property mask found!  Parsing value!
INFO: Property port found!  Parsing value!
INFO: Used configuration:
INFO: IP Address:  196.168.0.1
INFO: Mask:  255.255.255.0
INFO: Port:  2222
INFO: Saving configuration...
INFO: Configuration was saved!
INFO: Channel /dev/p_a_chan_dp_1 opened!
INFO: Channel /dev/p_a_chan_dp_2 opened!
INFO: Channel /dev/p_a_chan_pa opened!
...
```

Figure 32: Example part of trace file

7.2.5 Communication protocol with PC application

The communication protocol is based on a client-server model, where the client sends a request and then expects some kind of response. The Profibus Analyzer specific communication protocol was designed to be used for the communication between the Sitara processor and the PC.

Just note here that the requests/responses use naming convention, where *xxx_RQ* request invokes *xxx_RSP* response.

Requests

The client request contains at least three bytes - the length, command/data flag and command identification number (together they create a request header). The request header can be extended by data, which are specific for every request and will be described below. The byte order of the request header and additional data is shown in the table 33. Specific requests are shown in the table 1 altogether with a description below.

Figure 33: Request format

Description of fields in the request:

- *Length (LEN)* holds the total number of bytes which is equal to the header size plus data size.
- *CMD/DATA* holds the fixed value of 0xAA in case of the request (different value is used only in case of update request).
- *CMD_ID* is used to distinguish requests (commands).
- *DATA* holds request data (used if the command requires some parameter).

Command	LEN	CMD/DATA	CMD_ID	DATA
UPDATE_RQ	7	0xAA	0x31	4B
RESET_RQ	3	0xAA	0x24	No data
TERMINATE_RQ	3	0xAA	0x23	No data
START_CHANNEL_RQ	4	0xAA	0x22	1B
STOP_CHANNEL_RQ	4	0xAA	0x21	1B
SET_IP_RQ	13	0xAA	0x12	10B
SET_BDR_RQ	5	0xAA	0x11	2B
GET_STATUS_RQ	3	0xAA	0x04	No data
GET_ERR_DETAIL_RQ	3	0xAA	0x03	No data
GET_DEBUG_TRACES_RQ	3	0xAA	0x02	No data
GET_LOGGER_INFO_RQ	3	0xAA	0x01	No data

Table 1: Table of requests

Description of single requests:

- *UPDATE_RQ* updates firmware. It is described in the Section 7.5 in more detail.
- *RESET_RQ* invokes reboot of the Sitara processor. It is implemented to allow the user to reboot the Profibus Analyzer in a comfortable way without any manual operation.
- *TERMINATE_RQ* shuts down the Sitara processor. In order to boot it up again, it is necessary to switch power off and back on again.

- *START_CHANNEL_RQ* and *STOP_CHANNEL_RQ* starts/stops the selected Profibus Analyzer channel. The channel is selected by a single *DATA* byte (0x1 = channel DP1, 0x2 = channel DP2 and 0x3 = channel PA).
- *SET_IP_RQ* changes the Profibus Analyzer IP address, netmask and port. The first four *DATA* bytes contain the IP address in form *DATA[0].DATA[1].DATA[2].DATA[3]*, another four bytes contain the netmask in form *DATA[4].DATA[5].DATA[6].DATA[7]* and the last two bytes contain the port, which is computed as *DATA[8] * 256 + DATA[9]*.
- *SET_BDR_RQ* sets the baudrate of the selected Profibus Analyzer channel. The channel is selected in the same way as the one used in the start/stop channel request (contained in *DATA[0]*). The baudrate is contained in *DATA[1]*.
- *GET_STATUS_RQ* requests the status of the Profibus Analyzer (for more details see the response).
- *GET_ERR_DETAIL_RQ* requests the information about the last error of the Profibus Analyzer (for more details see the response).
- *GET_DEBUG_TRACES_RQ* requests traces saved in the Profibus Analyzer (for more details see the response).
- *GET_LOGGER_INFO_RQ* requests the information about individual versions of the Profibus Analyzer (for more details see the response).

Responses

The client response contains at least four bytes - the length, command/data flag, command identification number (together they create the response header) and response data. The byte order of the response is shown in the table 34. Specific responses are shown in the table 2 altogether with a description below.

Figure 34: Response format

Description of fields in the response:

- *Length (LEN)* holds the total number of bytes which is equal to the header size plus data size.
- *CMD/DATA* holds the fixed value of 0xAA.
- *CMD_ID* is used to distinguish responses.
- *DATA* holds response data.

Response	LEN	CMD/DATA	CMD_ID	DATA
UPDATE_RSP	4	0xAA	0x31	1B
RESET_RSP	4	0xAA	0x24	1B
TERMINATE_RSP	4	0xAA	0x23	1B
START_CHANNEL_RSP	4	0xAA	0x22	1B
STOP_CHANNEL_RSP	4	0xAA	0x21	1B
SET_IP_RSP	4	0xAA	0x12	1B
SET_BDR_RSP	4	0xAA	0x11	1B
GET_STATUS_RSP	7	0xAA	0x04	4B
GET_ERR_DETAILS_RSP	3 + data	0xAA	0x03	4B + strlen
GET_DEBUG_TRACES_RSP	0x55	0xAA	0x02	Trace file
GET_LOGGER_INFO_RSP	3 + data	0xAA	0x01	15B + strlen

Table 2: Table of responses

Description of single responses:

- *UPDATE_RSP* is more complicated. It is described in the Section 7.5 in more detail.
- Simple responses contain the single *DATA* byte which holds 0x1 if the request was successful and 0x2 if the request failed (in this case the application can request the last error or log to get the details). This includes
  - *RESET_RSP,*
  - *TERMINATE_RSP,*
  - *START_CHANNEL_RSP,*
  - *STOP_CHANNEL_RSP,*
  - *SET_BDR_RSP.*
- *SET_IP_RSP* contains the single *DATA* byte which holds 0x1 if the request was successful and 0x2 if the request failed. When the data are delivered and plausibility check is done, the response is sent from the old address. The address is then changed. In case of error during the IP address configuration, the old address is used again and therefore the application has to handle this.
- *GET_STATUS_RSP* contains four *DATA* bytes (see table 35) - the first one contains the information about opened channels (fields Chan. status), the information about running channels (fields Chan. run.) and Device status field, which indicates whether the application is running as expected. Another three bytes contain baudrates of all channels (fields Chan. speed).
- *GET_ERR_DETAILS_RSP* contains the response header and the string representation of last error which occurred in the application.
- *GET_DEBUG_TRACES_RSP* contains the response header and the string representation of the application trace file. This string is sent in 4kB transfers and terminated by a specific string.
- *GET_LOGGER_INFO_RSP* contains fixed fifteen *DATA* bytes and the string representation of kernel version (see table 36).

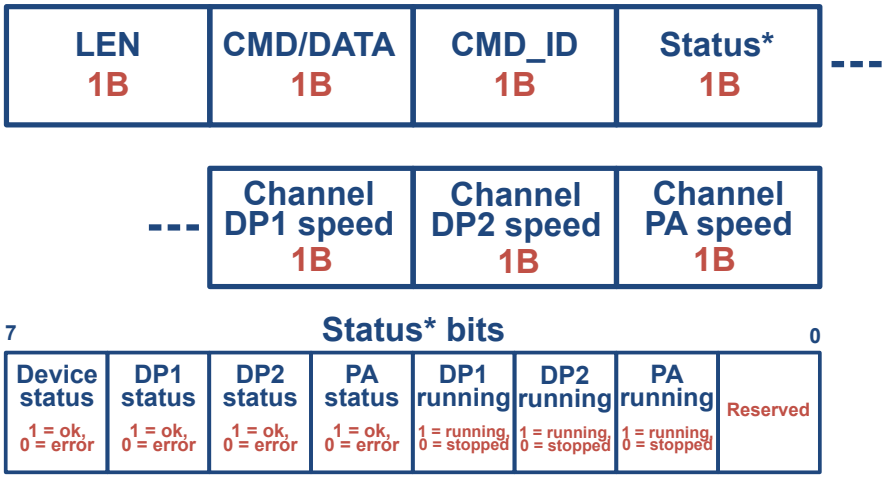


Figure 35: Get status request response

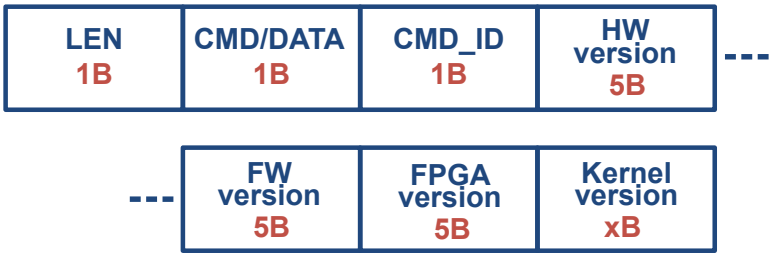


Figure 36: Get info request response

7.3 FPGA

The Profibus Analyzer Application and driver would be useless without the proper functionality of the FPGA. Because the FPGA design is not part of the thesis, the functionality will be just shortly described here. On the other hand, Sitara processor has to take care of FPGA bitstream loading, which is part of Profibus Analyzer Application and therefore is part of the thesis.

7.3.1 Design

The FPGA chip behaves like a static RAM (SRAM) and therefore, as mentioned before, the GPMC has to be configured correctly to match the timing hardcoded in the FPGA. Incoming Profibus data are stored in the FIFO which is accessed on the single address of SRAM. Additionally, there are a few configuration registers in the SRAM, which can disable/enable selected Profibus channel data capture, select the baudrate of the channel and so on.

The most important feature is the timestamp, which is added to data of all channels. It is essential to have the same time domain for all channels.

7.3.2 Bitstream loading

To be able to update the bitstream from the Sitara processor without any additional hardware connected (for example the Altera Blaster programmer), the FPGA is loaded in the Passive Serial Programming Mode.

Passive Serial Programming Mode

The Passive Serial Programming Mode is based on the usage of the SPI interface altogether with control signals. The host (Sitara) controls *nCONFIG*, *CONF_DONE* and *nSTATUS* signals and, of course, the SPI signals *DCLK* and *DATA*. In case of multiple FPGA chips are loaded by the host, the *nCE* pin can be used. On the other side, the FPGA can control *CONF_DONE* and *nSTATUS* signals. This is captured in the figure Figure 37 on page 42. [11]

The programming begins with the high-to-low pulse on the *nCONFIG* signal generated by the Sitara followed by high-to-low pulse on the *nSTATUS* signal generated by the FPGA. The *CONF_DONE* signal is pulled low by the FPGA and then the *nCONFIG* low-to-high pulse can be done by the Sitara followed by low-to-high pulse of the *nSTATUS* signal, which is released by the FPGA. At this moment, the SPI transfer of the bitstream in a correct format (which will be described in a paragraph below) can begin. After all the data are successfully transferred, the *CONF_DONE* pin should be released by the FPGA and therefore the low-to-high pulse should occur. Additional clocks (at least two) have to occur on the *DCLK* line. Then the *INIT_DONE* pulse, which should be pulled down by the FPGA during the data transfer, should be released back to the high state. This signalizes the end of the FPGA configuration and the start of the programmed function. This process is captured by a waveform in the figure Figure 38 on page 43. [11]

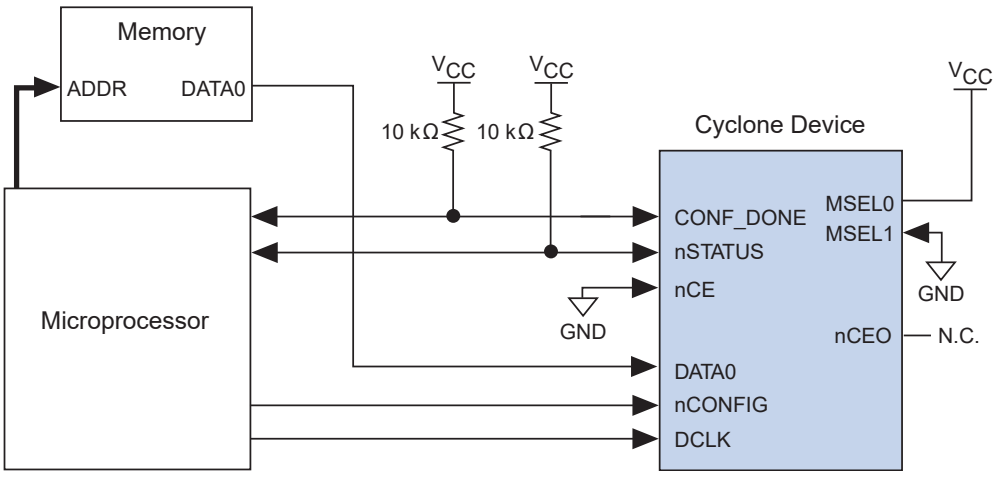


Figure 37: FPGA connection [11]

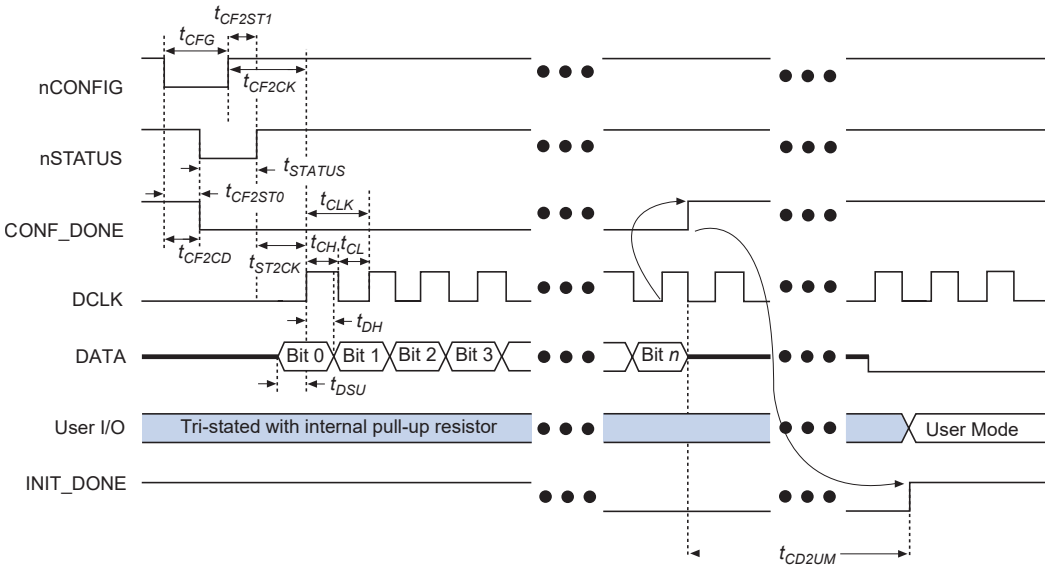


Figure 38: FPGA Passive Serial Programming mode waveform [11]

Bitstream file format

The bitstream is loaded in an RBF (Raw Binary File) format (which is the binary equivalent of TTF file). The RBF file can be generated directly by the Quartus software or by additional Quartus tools such as quartus_cpf, which can convert the bitstream formats. [11, 16]

7.4 Validation of application implementation

The verification process included the special FPGA bitstream used also for the Profibus Analyzer Driver function verification (see Subsection 6.4) and the test PC application.

7.4.1 PC test application design

The client PC application was implemented for test purposes (but it was also used to implement the PC library for communication with the Profibus Analyzer Application).

This test application allows to use all commands described above (the user can choose the command and parameters in simple command line menu) and also allows to test the data flow. Responses from the Sitara were displayed on the console to be checked manually.

The test bitstream in FPGA generates the incrementing sequence and therefore checking whether all data are coming correctly is easy. The long-term test was performed and all data were received correctly.

The test application also supports the test of the update protocol, which is described below in the Section 7.5. The update archive contained the testing script, which writes “Hello world” string to the file hello_world.txt. Therefore, after the update was performed, the .txt file was present in the Profibus Analyzer directory in Sitara processor indicating successful operation (this approach was used because the archive and temporary files generated during the update are deleted after the operation).

The test application was written in C/C++ and supports both Linux and Windows operating systems.

7.5 Firmware update strategy

This chapter describes the way how the firmware is updated in the Sitara processor by the Profibus Analyzer Application through the command from the Control PC Application. The command itself is described above in the Section 7.2.5.

7.5.1 Design

The update strategy was one of the important requirements during the development. The update feature has to be present in order to have a device, which is able to work many years without changing the hardware.

Several different ways of how to perform the update were discussed during the development. First, and the worst at the same, was to have separate commands for each update - firmware, application and kernel. This approach was denied because of no versatility. The second idea was to use some of existing open source update solutions but this was denied too because of the third party software with no functionality warranty.

The third idea, and finally the chosen one, was to have just a single update command which will download the update archive from the PC, decompress it and then just invoke the update script, which will do all the work. This approach is really simple and, what is important, it is also versatile (the script can do almost everything that is needed).

7.5.2 Update protocol

The update protocol was implemented according to the selected design. The update archive is transferred to the Profibus Analyzer and the update script will be invoked. To achieve at least the permissible level of safety (which in this case means the detection of error during the update process and reporting it to the control application), the update protocol contains several steps where the error can be reported (for the overview see the figure 39).

When the upgrade protocol is invoked, there is no way of recognizing other commands by the Profibus Analyzer. Because of this limitation, the update protocol is only allowed when all Profibus Analyzer channels are stopped. After the update is done, reset should follow, but during the development, this step was removed and replaced by more general command *RESET_RQ*, which can be sent from the PC control application after the update is done (or the Profibus Analyzer can still work with the old version and reset can be performed later, it is up to control application, which can ask user for decision).

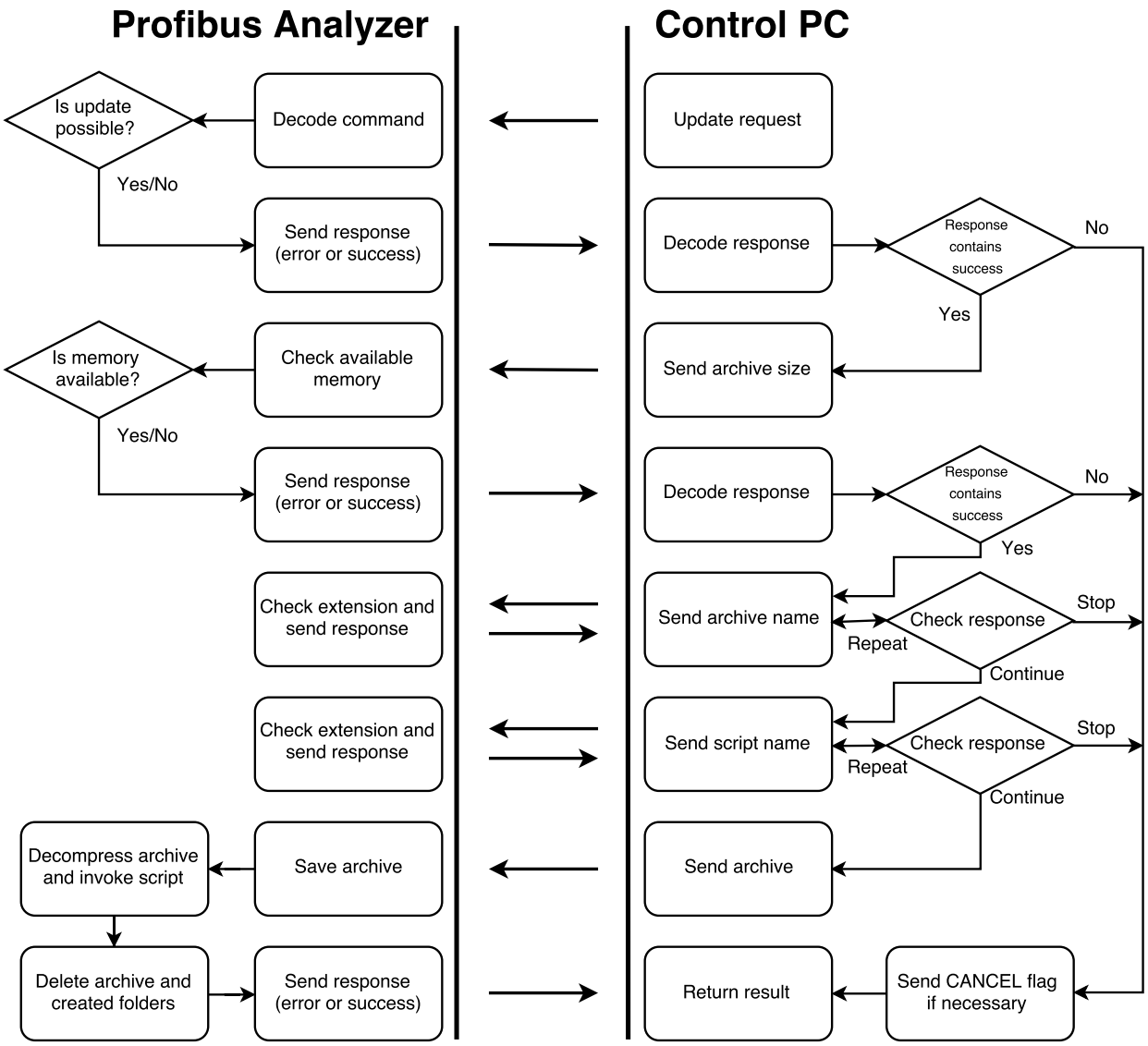


Figure 39: Update protocol

In this subsection, all steps from the figure 39 are described in more detail.

Responses from Profibus Analyzer Application

During the whole update process, responses have to be sent to the control application after each step. All of these responses have the same format, which is described by the table 40. The *STATUS* field contains 0x1 in case of success and 0x2 in case of error, other fields have fixed values.

LEN	CMD/DATA	CMD_ID	STATUS
1B	1B	1B	1B
0x4	0xAA	0x31	0x1/0x2

Figure 40: Update protocol response format

Update request

The update process begins with the request from the control PC (described in Section 7.2.5). When the command is decoded successfully by the Profibus Analyzer Application, some conditions have to be met for update process to be allowed, for example all channels have to be stopped and update has to be enabled by define during the compilation of the Profibus Analyzer Application. If these conditions are met, the response which contains the success is sent to the control PC, otherwise the error is contained and update process is terminated.

Archive file size detection

The update process can continue only if the previous response contains success. Otherwise, the update request has to be sent again after the blocking conditions are fixed (these can be detected by the last error reading request). The next step is a transfer of size of the update archive. This is necessary because there is a limited amount of memory in the Profibus Analyzer and in case of large archive the memory allocation can fail. To avoid this, the check of free memory is performed before the file itself is transferred. Therefore, the control application has to send a memory allocation request and wait for the response. If there is enough memory, the response contains success, otherwise the response contains error and the update process is terminated.

The size can be parsed as shown in figure 41, the memory allocation request format is shown in the figure 42. The new field contained in the request is a *FLAG* field, which contains the value 0x2 if the control application wants to terminate the update protocol (otherwise it contains the value 0x1 to continue). This will apply also for the rest of the protocol description.

```
unsigned int file_size = (unsigned int) (
          (buffer[P_A_DATA_OFFS + 0] << 0) |
          (buffer[P_A_DATA_OFFS + 1] << 8) |
          (buffer[P_A_DATA_OFFS + 2] << 16) |
          (buffer[P_A_DATA_OFFS + 3] << 24));
```

Figure 41: Size parsing from request

Figure 42: Size allocation request

Archive and script names

When the memory allocation is successfully done, the archive name and the update script name have to be transferred (the archive name first, then the script name). When the name is transferred, the extension (.tar.gz or .sh) and prefix (p_a_) is checked and the response is sent. This part is repeated until the valid string is entered or until the *FLAG* contains the continue flag (value 0x1), as it is described above.

Figure 43: Name request

Update archive transfer

After names of the archive and the script are successfully received (and the archive was created by the Profibus Analyzer Application), the update archive itself is transferred. The file is splitted to 4kB parts and sent to the Profibus Analyzer which saves received data to the file immediately (this approach was implemented to avoid the allocation of the buffer as large as the whole archive). When the whole file is sent, the file termination flag is sent from the control application for the Profibus Analyzer Application to be able to detect the end of file.

Figure 44: Archive transfer

Updating

The decompress and extract operation is performed on the received archive, the script is invoked and the return value from this script is checked. All received files are deleted and the response is sent to the control application.

7.5.3 Archive format

The update archive has to be in the appropriate format (.tar.gz). After the archive decompression, a folder named exactly the same as the archive (without the extension, of course) has to be created. To achieve that, the archive should contain a folder with the same name. The update script has to be placed in this folder altogether with the rest of files which are needed to perform the update. Both the update archive and the update script have to have p_a_ prefix and correct extension (see figure 45).

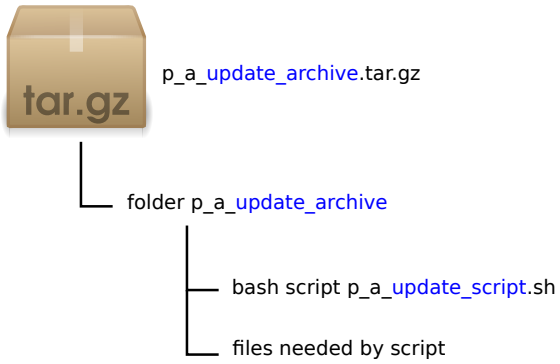


Figure 45: Archive structure

7.5.4 Usage

This approach can be used for all kinds of updates, for example:

- FPGA bitstream update - the archive will contain the bitstream file and the script will just remove the old one and replace it with the bitstream from the archive.
- Profibus Analyzer Driver update - the archive will contain either cross compiled files or source files together with the makefile. The script will then perform the compilation and installation of kernel modules.
- Profibus Analyzer Application update - the archive will contain either cross compiled files or source files together with the makefile. The script will then perform the compilation and will replace the old application.

8 Profibus Analyzer DLL

Although the PC application itself is not part of this thesis, the Dynamic Link Library was created within this thesis to support PC communication with the Profibus Analyzer device. This library will be used by final PC application and therefore the application does not have to take care about how the communication with the device works.

8.1 Dynamic Link Library (DLL)

A DLL is a library, which is loaded by the operating system (Windows or OS/2) into the context of an existing process. The Dynamic Link Library file has extension .dll.

The main advantage of the DLL is that it can be used by many programs at the same time. The example of this feature is really simple - many functionalities of the operating system are provided by the DLLs. DLLs is a great way to have efficient memory usage, code reuse and modular code architecture (much of the functionality of an application can be provided by DLLs). Also, the update is really simple, the DLL is replaced by new one and there is no need to recompile the program.

8.1.1 Introduction to DLL

There are two different approaches of how the DLL can be used [18]:

- Load-time dynamic linking means that the DLLs have to be already present in the memory when the application is loaded. The usage in the code is the same as it would be the local function. During the compile time the linker will not insert the code of DLL function, but it saves the reference which says where the function is located. Before the application can run, the loader has to find all references to DLL functions and replace them with the entry points. If any reference is not present, the application cannot run.
- Run-time dynamic linking means that the DLLs does not have to be present in the memory when the application is loaded. They will be loaded on demand by the LoadLibrary (or LoadLibraryEx) at run-time when it is requested by the application. When the library is loaded, the handle of DLL is returned and by using the GetProcAddress the appropriate function (pointer to function) in the DLL can be found. The application can run although the requested DLL cannot be loaded and, in worst case, it can at least print error message.

The standard Windows DLL is written in the C++ language (or in C, of course), which is extended by additional features required for the DLL development.

The first difference from standard C++ language is that it contains exported functions, which can be called by the application. These functions are declared by the ___declspec(dllexport) define.

Another extension is the usage of entry-point function. This function is invoked every time the process (or thread) is attached (detached) to the DLL. The actual state (attach/detach thread/process) is passed to this function by one of the arguments.

8.1.2 C++/CLI

Special case can occur, when the DLL (unmanaged code) is called by the managed code (for example C#).

Managed/unmanaged code

It is worth to provide a simple explanation of managed/unmanaged code here:

- Managed code is managed by the Common Language Runtime (CLR). The code is compiled into bytecode which is called Common Intermediate Language (CLI). The CLI is converted to native (machine specific instructions) code by the CLR. The CLR provides a lot of additional services to the application. For example memory management, exception handling, security, thread management and garbage collection. The CLI code can run on any machine where the CLR is present and therefore it is portable.
- Unmanaged code does not need any runtime environment. It is compiled directly to the machine specific instructions and therefore it is not portable. The memory management is handled directly by the application and size of types is defined by the compiler during runtime.

Therefore, to connect these two different programming language worlds, the C++/CLI was introduced by the Microsoft. The purpose is to enable the usage of Windows DLLs written in unmanaged C++ by the managed code, for example by the GUI application written in C#.

C++/CLI language

C++/CLI is not considered to be just an extension of the C++, but a new language of its own (although it uses a lot of C++ functionality). As mentioned before, the C++/CLI was introduced by the Microsoft to enable usage of unmanaged code from the managed code and vice versa (the reader can imagine it like a bridge between managed and unmanaged code). It is meant to be the replacement for the deprecated Managed Extensions for C++, which was used for the same purpose. C++/CLI has its own, for example character * is no longer used for pointers, the ^ character is used instead.

The C++/CLI language is supported in Visual Studio. The only way of how to compile and debug C++/CLI is to have Microsoft Visual C++ IDE installed on the host machine. Therefore, the C++/CLI is not portable to another operating system, for example Linux. [19]

Marshaling

Marshaling is simply the data transformation from one format to another. It is usually considered to be the same process as serialization. A simple example is data transfer, when the memory representation of some object is transformed to another format suitable for transmission and then transferred.

C++/CLI has built-in support for marshaling between managed and unmanaged types (for passing data between C++ and C#). This feature is essential for C++/CLI purpose.

The standard primitive data types, for example int, double or bool, are not affected by this. On the other side, complex types such as strings are affected and have to be marshalled. [19]

Wrapper

In this case the wrapper means the middleware code between the Windows DLL and the C# application. This middleware is written in the C++/CLI. It encapsulates the functions of the DLL and provides them to the managed code application.

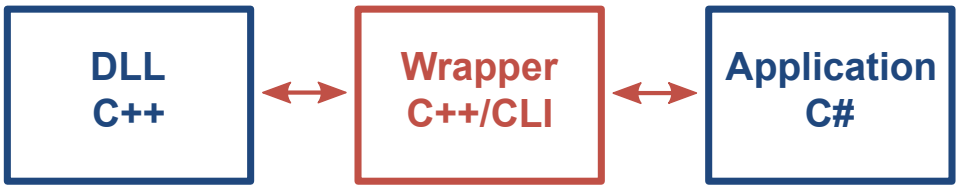


Figure 46: Wrapper architecture

8.2 Profibus Analyzer DLL

The Profibus Analyzer DLL will encapsulate whole communication to/from Profibus Analyzer device including the decoding of received Profibus telegrams.

8.2.1 Design

Profibus Analyzer Dynamic Link Library was designed to support usage from both C++ (unmanaged) and C# (managed) code. This is possible because the C++/CLI wrapper was also designed.

To achieve the maximal usability, UDP packets sent from Profibus Analyzer are captured by using the Windows Packet Capture library. This is necessary because of possible security settings of the corporate network, where UPD packets can be dropped by Windows OS.

When the library is used from C#, the wrapper is compiled as DLL and Profibus Analyzer DLL is no longer compiled as DLL but as a static library and the macro *COMPILE_AS_DLL* has to be commented.

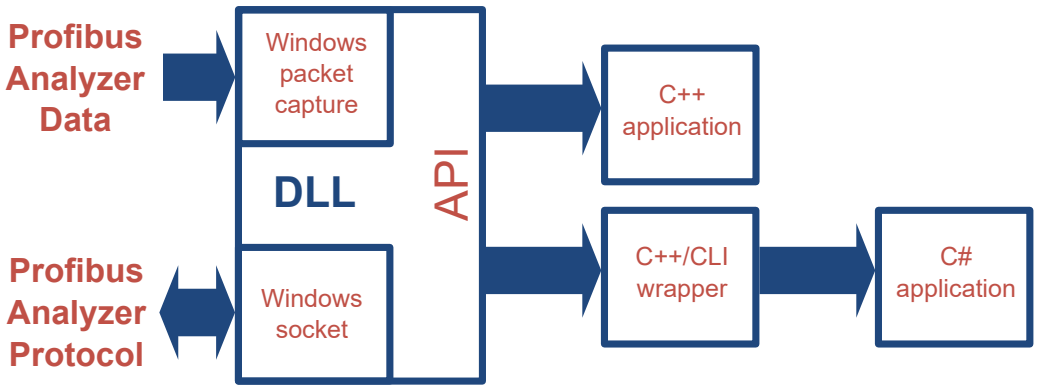


Figure 47: DLL architecture

The DLL provides a single object (class) called ProfibusAnalyzer, which encapsulates the whole communication. Every single command has its own function. After the command is sent, the response is expected immediately and therefore every function is a blocking call.

To provide data in a non-blocking way, the poll approach was implemented. Function *Channel_data_available* will return true (boolean) when some data are available and then the function *Get_channel_data* will return an array of raw data from specified Profibus Analyzer channel. When the channel is started, DLL creates a new thread where packets from Profibus Analyzer are continuously captured.

Windows Packet Capture library

The WinPcap (short for Windows Packet Capture) is an open source library, which provides a programming interface for capturing raw data from network packets for Windows OS. It is widely used, for example by the Wireshark, which is a famous tool for network diagnostics. [17]

It allows creating a packet filter that is used to specify packets which are requested by the application. The filter can be used for example to capture only UDP packets on the specified port from specified IP address. This is an important feature which is used by Profibus Analyzer DLL to get raw data packets sent by Profibus Analyzer application.

8.2.2 Provided API

The provided API is almost similar for the C++ code and for the C# code. There are only small differences because of additional wrapper layer.

The library is written in form of a class called ProfibusAnalyzer, which has these public functions (all functions instead of constructor return boolean, to keep list clear it is not written before the name):

- *ProfibusAnalyzer(void)* is the constructor of Profibus Analyzer object. It performs only initialization of internal structures of class.
- *Init(pa_address_t addr)* initializes the connection to the Profibus Analyzer device, which has to be present on selected IP address.
- *Deinit(void)* closes sockets opened by Init function.
- *Reboot(void)/Poweroff(void)* functions are used to send the command to reboot/poweroff Sitara processor.
- *Set_IP_address(pa_address_t*[2] *addr)* changes the address of Profibus Analyzer. This address is then stored in memory and used by default after device startup.
- *Get_status(pa_status_t**[3] *status)/Get_info(pa_info_t* info)* functions are used to get Profibus Analyzer runtime status and information about firmware/hardware/fpga/kernel versions.
- *Set_baudrate(pa_channel_t chan, pa_baudrate_t speed)* selects speed of Profibus channel connected to Profibus Analyzer.
- *Start_channel(pa_channel_t chan)/Stop_channel(pa_channel_t chan)* functions are used to start or stop selected Profibus Analyzer channel.
- *Channel_data_available(pa_channel_t chan)* returns true if some data are available for selected channel.
- *Get_channel_data(pa_channel_t chan, uint8_t**[4] *data, int* len)* saves available data of selected channel into the buffer.
- *Perform_update(char**[5] *archive_name, char* script_name, uint8* archive_data, unsigned int archive_len)* initialize update protocol, transfers update archive and checks result of the operation.
- *Get_adapter_list(nwk_adapter_list_t* l, int max)* is necessary for correct WinPcap functionality, because it has to know which network adapter should be used for packets capturing. This function returns a list of network adapters and by using function Set_adapter(int idx) the user selects which adapter is used. The data transfer from Profibus Analyzer will not work until correct network adapter is set.

[2]Names of structures are in case of C# code extended by 'wrapper' string. For example pa_address_t will be used from C++ and pa_address_wrapper_t will be used from C#.

[3]Pointers are in case of C# changed to references.

[4]uint8_t is in C# changed to Byte.

[5]char* is in C# changed to System::String reference.

8.3 Validation of DLL implementation

The verification process included again the special FPGA bitstream, which was used before to test Profibus Analyzer Driver and Application and the test C++ and C# applications.

8.3.1 Test applications design

The steps of verification were almost the same as in case of the Profibus Analyzer Application, which can be found in the Section 7.4. The C++ and C# applications were written to test every function of ProfibusAnalyzer class including the data transmission (the incrementing sequence was checked by the top-level C++ and C# application).

9 Conclusion

The aim of this thesis was to implement a firmware for the Profibus Analyzer device, which will run on the Sitara processor. While the thesis was being written, requirements were changed and therefore also the DLL, which encapsulates the communication between PC and the Profibus Analyzer itself, was added to requirements for this thesis.

The first thing which had to be done, was to run Linux Operating System on the Profibus Analyzer device, or in more detail on the Sitara processor. The processor Linux SDK from the Texas Instruments was used and the RT-Linux was successfully ported to the Profibus Analyzer hardware.

The next step was the implementation of the Profibus Analyzer Driver. The driver controls the data transfer from the FPGA to the DDR3 memory, which is done automatically by the Enhanced DMA peripheral. Every byte transfer is triggered by the DMA event pins, which are controlled by the FPGA. The driver also passes the information about the data buffer to the user-space application. Only buffer address and size are passed to the user-space application to avoid unnecessary data copying, and to limit the CPU overhead.

To complete the data transfer from the FPGA to the PC via the Ethernet interface, the Profibus Analyzer Application was implemented. It receives the information about the buffer position and data size from the Profibus Analyzer Driver and immediately sends the data in form of the UDP packet to the PC. This application also ensures the control communication with the PC and therefore supports its own communication protocol. This protocol contains several control commands, for example enable/disable Profibus Analyzer channel, set specific channel speed, and get error/info log.

The last part that has to be supported by the Sitara processor, was the implementation of the update strategy. The simplest solution was adding the update command to the already existing communication protocol. The final approach is to transfer .tar.gz archive, which contains all files necessary for the update, from the PC to the Sitara processor together with the name of update script. The Profibus Analyzer Application performs decompressing of archive (untar) and invokes the update script. After the script finishes, it deletes all created files. This ensures all kinds of actions to be done with a single update command, for example FPGA firmware update, Profibus Analyzer Driver update or Profibus Analyzer Application update.

On the PC side it was necessary to implement DLL which encapsulated all communication from/to Profibus Analyzer device and therefore the PC application itself does not take care about that and it can focus only on the GUI. The DLL also contains decoder, which decodes Profibus telegrams from received UDP packets to provide full service to the GUI application.

Finally, all pieces of software described above were tested in a real situation. All goals and requirements were fulfilled and solution works as required.

Problems

During the development, there were many problems that had to be solved. First, and in my opinion the worst, problem occurred right at the beginning with the Linux porting to Profibus Analyzer hardware. Because of no previous experience with this topic, it took a lot of time to get Linux running on the board. Problems were caused by the different DDR3 memory (timing has to be configured) and the lack of SD card detecting signal, which was not connected to the Sitara processor. These problems were solved by writing our own board.c file and by changing few lines in SD card controller driver.

Another problem was with the GPMC configuration and therefore the FPGA access. The problem was solved by correct timing and multiplexing configuration in the device tree and by updating the FPGA bitstream that had a bug in the memory access controller.

Because the Profibus Analyzer will behave as a server with the static IP address, the IP address has to be configured automatically during the boot. The problem was that standard ways of how to achieve

that (for example change the /etc/network/interfaces file) did not work. Therefore the initialization script that changes IP address by command ifconfig was installed to run automatically.

The last problem mentioned here is a problem with the DMA transfer. The Profibus Analyzer Driver needs to know how many bytes have been transferred so far by the DMA transfer, but this information is not provided by the DMA driver. Because of this limitation, the manual access to the DMA PaRAM has to be performed and the number of transferred bytes is found directly in the appropriate transfer by the Profibus Analyzer Driver.

To be improved

There is, as always, something to be improved, to be done in another, better, way. In our case it is the initialization of Profibus Analyzer Application. When something goes wrong (for example the bitstream loading fails), there is no functionality which reports this error to the PC application and no way of how to fix this situation from the PC application. The only way of how to handle this is to connect to the board manually and see the error log in the console. This approach should be improved to handle these rare situations when something unexpectedly fails.

References

[1] FELSER, Max. The Fieldbus Standards: History and Structures [online]. Bern, 2002 [cit. 2018-03-02]. Article. Bern University of Applied Sciences.

[2] ZURAWSKI, Richard. The industrial communication technology handbook. Boca Raton, Fla.: Taylor & Francis, c2005. ISBN 0849330777.

[3] PROFIBUS System Description: Technology and Application [online]. Karlsruhe, Germany: PROFIBUS Nutzerorganisation, April 2016 [cit. 2018-03-15]. Available from: https://www.profibus.com/download/profibus-technology-and-application-system-description/

[4] SOFTING, Protocol Analyzer for PROFIBUS DP and PA: Detailed Analysis of Bus Communication. Softing Industrial Automation 2012.

[5] BOVET, Daniel Pierre a Marco CESATI. Understanding the Linux Kernel. 3rd ed. Sebastopol: O´Reilly, 2005. ISBN 0-596-00565-2.

[6] TORVALDS, Linus a David. DIAMOND. Just for fun: the story of an accidental revolutionary. New York, NY: HarperBusiness, c2001. ISBN 0066620724.

[7] LIU, Jane W. S. Real-Time systems. Upper Saddle River, NJ: Prentice Hall, c2000. ISBN 978-0130996510.

[8] MCKENNEY, Paul. A realtime preemption overview. LWN.net [online]. 10 August 2005 [cit. 2018-03-20]. Available from: https://lwn.net/Articles/146861/

[9] Atmel Corporation. Atmel | SMART ARM-b ased Embedded MPU datasheet. Revised 31 August 2015.

[10] Texas Instruments. AM335x and AMIC110 Sitara™ Processors Technical Reference Manual. October 2011, Revised March 2017.

[11] Altera. Cyclone Device Handbook, Volume 1. Revised May 2008.

[12] Linux Core U-Boot User's Guide [online]. Texas Instruments [cit. 2018-04-16]. Available from: http://processors.wiki.ti.com/index.php/Linux_Core_U-Boot_User%27s_Guide

[13] SALZMAN, Peter Jay, Michael BURIAN and Ori POMERANTZ. The Linux Kernel Module Programming Guide. The Linux Documentation Project [online]. 2001, 18 May 2007 [cit. 2018-04-16]. Available from: http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html

[14] Acromag, Inc. BusWorks TM 900PB Series ProfiBus/RS485 Network I/O Modules Technical Reference, INTRODUCTION TO PROFIBUS DP. [cit. 2018-03-15]. Available from: http://www.diit.unict.it/users/scava/dispense/II/Profibus.pdf

[15] Processor SDK Linux Training: Introduction to Device Driver Development [online]. Texas Instruments [cit. 2018-04-21]. Available from: http://processors.wiki.ti.com/index.php/Processor_SDK_Linux_Training:_Introduction_to_Device_Driver_Development

[16] Raw Binary File (.rbf). Quartus II Help v14.1 [online]. Altera [cit. 2018-04-23]. Available from: http://quartushelp.altera.com/14.1/mergedProjects/reference/glossary/def_rbf.htm

[17] WinPcap Documentation. WinPcap Documentation [online]. The WinPcap Team [cit. 2018-05-02]. Available from: https://www.winpcap.org/docs/docs_412/html/main.html

[18] Dynamic-Link Libraries. Microsoft [online]. Microsoft [cit. 2018-05-02]. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx

[19] .NET Programming with C++/CLI (Visual C++). Microsoft [online]. Microsoft [cit. 2018-05-10]. Available from: https://msdn.microsoft.com/en-us/library/68td296t.aspx

List of Figures

List of Tables

Appendix

GPMC settings

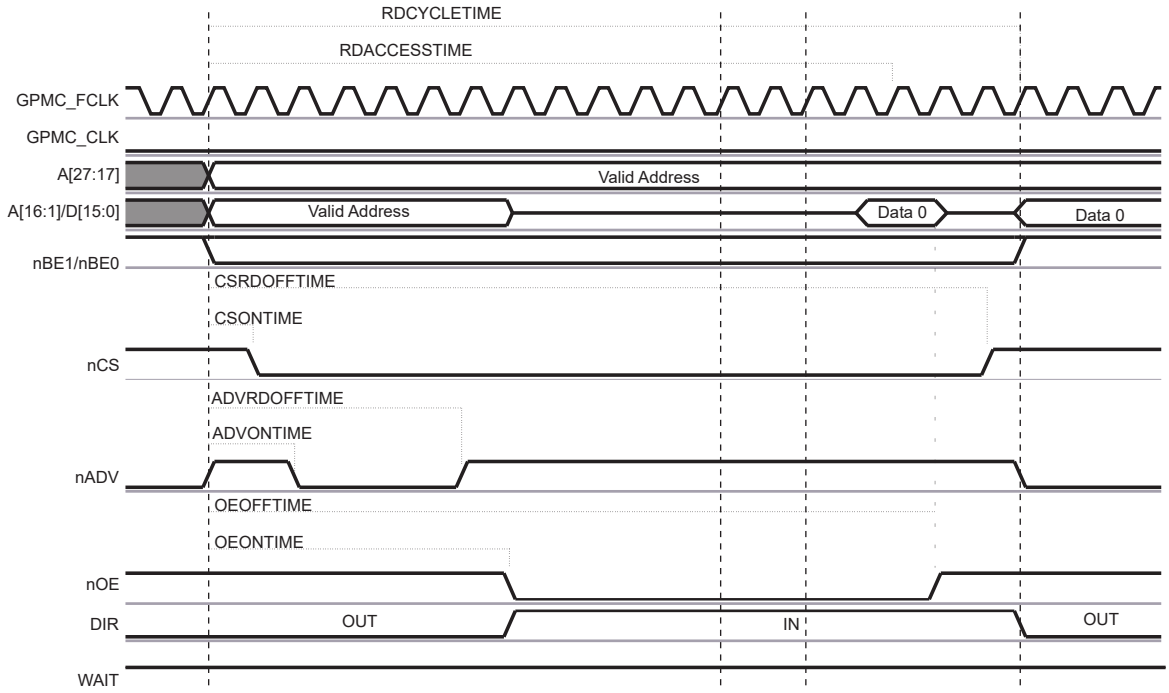


Figure 48: Asynchronous Single Read Operation on an Address/Data Multiplexed Device [10]

```
gpmc_pins_default:  gpmc_pins_default {
     pinctrl-single,pins = <
          AM33XX_IOPAD(0x800, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad0.gpmc_ad0 */
          AM33XX_IOPAD(0x804, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad1.gpmc_ad1 */
          AM33XX_IOPAD(0x808, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad2.gpmc_ad2 */
          AM33XX_IOPAD(0x80c, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad3.gpmc_ad3 */
          AM33XX_IOPAD(0x810, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad4.gpmc_ad4 */
          AM33XX_IOPAD(0x814, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad5.gpmc_ad5 */
          AM33XX_IOPAD(0x818, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad6.gpmc_ad6 */
          AM33XX_IOPAD(0x81c, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad7.gpmc_ad7 */
          AM33XX_IOPAD(0x820, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad8.gpmc_ad8 */
          AM33XX_IOPAD(0x824, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad9.gpmc_ad9 */
          AM33XX_IOPAD(0x828, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad10.gpmc_ad10 */
          AM33XX_IOPAD(0x82C, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad11.gpmc_ad11 */
          AM33XX_IOPAD(0x830, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad12.gpmc_ad12 */
          AM33XX_IOPAD(0x834, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad13.gpmc_ad13 */
          AM33XX_IOPAD(0x838, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad14.gpmc_ad14 */
          AM33XX_IOPAD(0x83C, PIN_INPUT_PULLUP | MUX_MODE0) /* gpmc_ad15.gpmc_ad15 */
          AM33XX_IOPAD(0x870, PIN_INPUT | MUX_MODE0)    /* gpmc_wait0.gpmc_wait0 */
          AM33XX_IOPAD(0x874, PIN_OUTPUT | MUX_MODE0)   /* gpmc_wpn.gpmc_wpn */
          AM33XX_IOPAD(0x87c, PIN_OUTPUT | MUX_MODE0)   /* gpmc_csn0.gpmc_csn0 */
          AM33XX_IOPAD(0x880, PIN_OUTPUT | MUX_MODE0)   /* gpmc_csn1.gpmc_csn1 */
          AM33XX_IOPAD(0x890, PIN_OUTPUT | MUX_MODE0)   /* gpmc_advn_ale.gpmc_advn_ale */
          AM33XX_IOPAD(0x894, PIN_OUTPUT | MUX_MODE0)   /* gpmc_oen_ren.gpmc_oen_ren */
          AM33XX_IOPAD(0x898, PIN_OUTPUT | MUX_MODE0)   /* gpmc_wen.gpmc_wen */
          AM33XX_IOPAD(0x89c, PIN_OUTPUT | MUX_MODE0)   /* gpmc_be0n_cle.gpmc_be0n_cle */
     >;
};
```

```
&gpmc {
      status = "okay";
      pinctrl-names = "default";
      pinctrl-0 = <&gpmc_pins_default>;
      #address-cells = <2>;
      #size-cells = <1>;

      /* start address is 0x02000000, size is 0x01000000 - 16MB */
      ranges = <1 0 0x02000000 0x01000000>; /* CS1:  FPGA */

      fpga@1,0 {
            reg = <1 0 0x01000000>;
            bank-width = <2>;
            gpmc,mux-add-data = <2>; /* 1 = address-address-data */
            /* 2 = address-data */
            /* Config 2 */
            gpmc,cs-on-ns = <40>; /* max 15 ticks */
            gpmc,cs-rd-off-ns = <440>; /* max 31 ticks */
            gpmc,cs-wr-off-ns = <440>; /* max 31 ticks */
            /* Config 3 */
            gpmc,adv-on-ns = <80>; /* max 15 ticks */
            gpmc,adv-rd-off-ns = <160>; /* max 31 ticks */
            gpmc,adv-wr-off-ns = <120>; /* max 31 ticks */
            /* Config 4 */
            gpmc,oe-on-ns = <200>; /* max 15 ticks */
            gpmc,oe-off-ns = <400>; /* max 31 ticks */
            gpmc,we-on-ns = <200>; /* max 15 ticks */
            gpmc,we-off-ns = <400>; /* max 31 ticks */
            /* Config 5 */
            gpmc,rd-cycle-ns = <500>; /* max 31 ticks */
            gpmc,wr-cycle-ns = <500>; /* max 31 ticks */
            gpmc,access-ns = <360>; /* max 31 ticks */
            gpmc,page-burst-access-ns = <300>; /* max 15 ticks */
            /* Config 6 */
            gpmc,bus-turnaround-ns = <300>; /* max 15 ticks */
            gpmc,cycle2cycle-delay-ns = <300>; /* max 15 ticks */
            gpmc,wr-data-mux-bus-ns = <160>; /* max 15 ticks */
            gpmc,wr-access-ns = <600>; /* max 32 ticks */
            gpmc,cycle2cycle-samecsen;
            gpmc,cycle2cycle-diffcsen;
      };
};
```

Content of CD

- PDF - diploma thesis
- Source codes divided into following folders:
  - PA_Driver contains source codes of developed kernel module.
  - PA_Application contains source codes of developed Linux application.
  - PA_DLL contains source codes of developed Dynamic Link Library together with C++/CLI wrapper. This folder also contains examples how to use both the DLL (from C++) and wrapper (from C#).