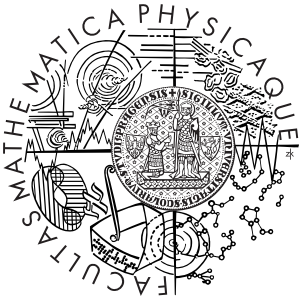
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Bc. Gabriel Zábušek

Bus monitoring and control environment

Department of Software Engineering

Thesis supervisor: RNDr. David Obdržálek
Study program: Software Systems

2010

I would like to thank to:

RNDr. David Obdržálek
My family
The team of the Faculty of mathematics and physics at the Charles University

Prohlašuji, Že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Gabriel Zábušek

Contents

Title: Bus monitoring and control environment
Author: Bc. Gabriel Zábušek
Department: Department of Software Engineering
Supervisor: RNDr. David Obdržálek
Supervisor's e-mail address: David.Obdrzalek@mff.cuni.cz

Abstract:

This text is intended as an accompaniment to software and hardware environment for monitoring and controlling digital buses named Bus-Spy. The thesis analyses the problems of bus control, bus monitoring and existing solutions. It offers a solution in form of a very modular hardware/software environment which is capable of complete monitoring and controlling the i2c (Inter Integrated Circuit) bus and the CAN (Controller Area Network) bus in their basic form. It also explains the software and hardware design of the environment and describes the implementation.

The thesis should mainly serve low level and system developers, firmware engineers or anyone involved with digital bus related communication.

Keywords: bus, environment, emulation, control, simulation, monitoring, analysis

Název práce: Bus monitoring and control environment
Autor: Bc. Gabriel Zábušek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. David Obdržálek
e-mail vedoucího: David.Obdrzalek@mff.cuni.cz

Abstract:

Tento text je především určen jako doprovod k softwarovému a hardwarovému prostředí na monitorování a kontrolu digitálních sběrnic. Tato práce analyzuje problémy kontroly a monitorování digitálních sběrnic a řešení, která jsou na trhu k dispozici. Práce poskytuje řešení ve formě velice modulárního hw/sw prostředí které je schopno plně kontrolovat a monitorovat i2c a CAN sběrnice v jejich základní formě. Také je zde vysvětlen návrh softwaru a hardwaru tohoto prostředí a popis jejich implementace.

Práce je především určena nízkoúrovňovým programátorům, firmware inženýrům, nebo komukoliv pracujícímu s komunikací po digitálních sběrnicích.

Klíčová slova: sběrnice, prostředí, emulace, kontrola, simulace, monitorování, analýza

Chapter 1

Introduction

1.1 Description and purpose of this work

The purpose of this work is to create a software and hardware environment for (non intrusively) monitoring[1] and controlling[2] digital buses. The first version should be able to monitor and control the I2C (Inter-Integrated Circuit) bus and the CAN (Controller Area Network) bus. It should support at least the basic versions of these buses specified by the latest freely available specifications. In case the latest specification is non free I won't be able to guarantee support for such version.

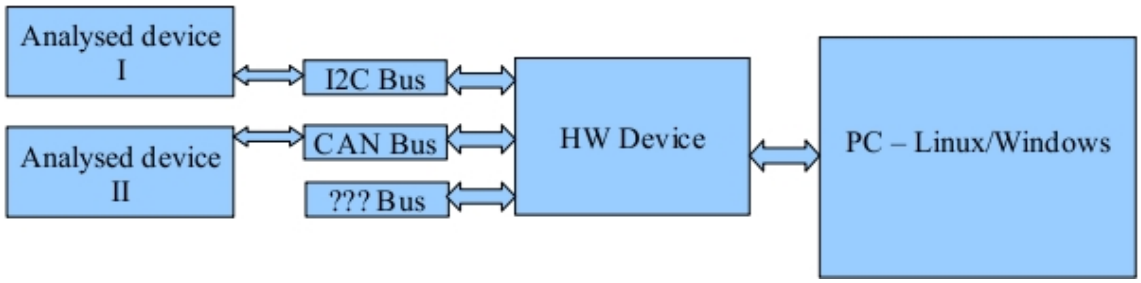


Figure 1.1: Basic structure of the environment

It will be designed in such a way that adding support for a new bus should be a matter of a short period of time. The whole project will consist of two main parts. A hardware device connected to the users PC and an application running on a personal computer powered by either Windows or Linux operating system.

The hardware device will be used to get the data from a bus[3] and transfer them to the PC or to receive data from the PC and send them onto the bus. Ideally most processing should be done on the PC site and the hardware device should be easily replicable. The software on the PC site will take care of the following (at least):

- Analysing the data received from the device.
- Displaying the data in a logically correct manner.
- Plotting the graphical data on a computer screen.

[1] Scanning and analysing data on the bus without influencing the devices connected to the bus
[2] For instance creating virtual devices or acting as a bus master
[3] based on the (electric) specification of some specific bus

- Sending previously defined data to the device.
- Offering a way to emulate protocols / devices manually or by an external script by the user.

This kind of an environment has a wide variety of usages in research and development practice. Just to name a few it could be used:

- During development of hardware or device drivers for a specific bus
- Diagnostics of a bus
- Reverse engineering
- and so on...

Chapter 2

Analysis

In this chapter I will analyse existing solutions similar to this work which are available, describe their pros and cons and analyse the possible approaches to successfully accomplish the requirements for this work.

2.1 Existing solutions

There are many solutions for analysing and controlling digital buses on the market today. More or less analysers for most of the existing standard computer buses (such as PCI, PCI-E, USB, CAN, SATA, SAS, SCSI etc.) are available. However, as far as I am concerned there is no robust universal solution available. Most of the tools on the market are usually targeted at one specific bus and are quite expensive. The prices of these devices vary from $300 up to $30,000 (and possibly more)! Usually they are sold as hardware / PC software couples and the hardware parts are quite complex. The following list summarizes the biggest issues of bus monitoring and controlling tools available on the market today:

- Targeted for one specific bus, No universal solution
- hardware/software couple needed for each digital bus
- High price
- Very complex and non-extensible hardware portion of the solution

In my work I would like to target all these issues by insisting on:

- Proper advanced software design
  - Portability
    - Support for several operating systems and architectures
  - Scalability
    - Adding support for new buses
    - Adding support for new hardware modules
    - Adding support for new connections of hardware modules with computer and therefore support more types of personal computers (laptops etc.)

  - Object Oriented
    - Use of design patterns
  - User-friendliness

- Simplified but highly scalable hardware portion of the environment

  - Supporting or giving the possibility to support large amount of digital bus types

In the following text I will list and analyse the solutions which I have found on the Internet. I consider several of them interesting and inspiring.

2.1.1 LeCroy VBA104Xi Vehicle Bus Analyser

Very advanced CAN bus analyser targeted for use in the automotive industry (CAN is used in automotive electronics very often). The short description on the producers website states:

"The Vehicle Bus Analyser decodes CAN serial data into Symbolic (application layer) text directly on the oscilloscope display. For the first time, an engineer has both the full range of CAN protocol stack information, symbolic, hex, and electrical signal and the ability to view additional in-circuit electrical signals (sensors and actuators, voltage levels, transients, etc.) that influence the CAN bus. In addition, up to four different CAN buses can be decoded at one time.Standard and specialized oscilloscope tools can be used to validate and debug designs."[24]

Ideally once my work is finished it will be possible to add all the functionality to offer as many functions to analyse any bus as this device does for CAN bus. However, in my opinion, this device is a nice example of the aforementioned problems. The main downsides of this product are these:

Figure 2.1: CAN bus analyser for the automotive industry

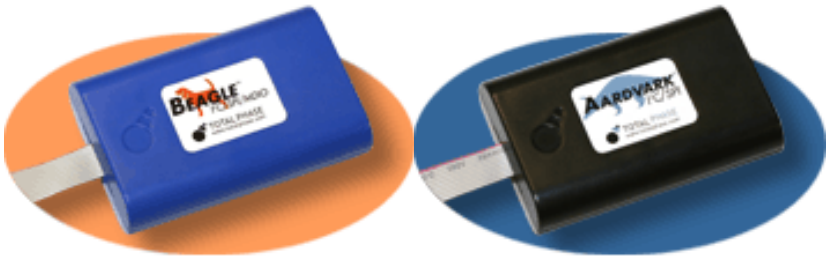
- Extreme price: $24,000
- It is a desktop computer itself
- Supports CAN bus only
- Doesn't offer any emulation/control features

The features which I found impressive or inspiring are the following:

- Highly advanced protocol analysis features
- Electrical properties analysis features
- Four different CAN buses analysis at the same time

2.1.2 Beagle I2C/SPI Protocol Analyser & Aardvark I2C/SPI Host Adapter

Set of several hardware and software tools for analysing and controlling I2C and SPI buses [25]. Overall this set of products is very similar to what I would like to accomplish in this work but ideally with possibility to support much larger scale of digital bus types. The software provided is free of charge and offers most of the functions I would like to provide in my solution. The following list summarizes what this set of tools offers and what I found inspiring:



(a) Analyzer (b) Host adapter

Figure 2.2: Beagle I2C/SPI Analyzer and Emulator

- Real-Time Data Capture and Display - Watch I2C, SPI, and MDIO packets as they occur on the bus
- Interactive filtering and searching in real time with LiveFilter™ and LiveSearch™ tools
- Packets data grouped in an expandable tree view format
- Non-intrusively monitor I2C up to 4 MHz
- Non-intrusively monitor SPI up to 24 MHz
- I2C master and slave up to 400 kHz
- I2C multi-master support
- SPI master up to 8 MHz
- SPI slave up to 4 MHz
- Fully Windows, Linux, and Mac OS X compatible
- Good scalable software free of charge

The main downsides of this set are the following:

- Two devices needed to properly analyse and control one bus
  - Analyser + Host Adapter
- "One device per bus" approach

- No electric characteristics analysis support
- The provided software is not open source
- Only USB connection with computer is supported

2.1.3 CAS-1000-I2C/E^{TM} Bus Analyser, Exerciser, Emulator, and Programmer

A very powerful I2C bus monitoring and controlling environment [26]. The inspiring functions what this device offers are listed here:

- Supports I2C and SMBus
- Easily connects to PCs and workstations via USB 2.0 interface
- Supports Standard-mode, Fast-mode, and Fast-mode Plus (Fm+) with I2C bit rates of up to 5 Mbit/sec
- Supports High-speed mode (Hs-mode) monitoring of up to 5 Mbit/sec
- Passive traffic monitoring/recording (both state and timing displayed) with time-stamping, message filtering, and symbolic translating
- Continuous logging of data to disk files
- Programmable trigger event to highlight bus transactions of interest and/or signal target or external instrument
- Active programmable and interactive traffic generation supporting all modes including Master, Multi-Master, and Slave while emulating up to 1 Master and/or up to 10 Slaves concurrently
- Forced I2C error injection can be programmed in the outgoing stream
- Programmable bus pull-up resistors on SDA and SCL lines ranging from 250 to 50K ohms (or floating when target pulls up)
- Scope support
- **Powerful command/script language for testing/emulation control**

The main downsides of this solution are the following:

- Software running under Windows only
- Only I2C SMBus supported
- Price: $8,000

The following list of illustrations (2.3, 2.4, 2.5, 2.6) shows the main functions of the provided software.

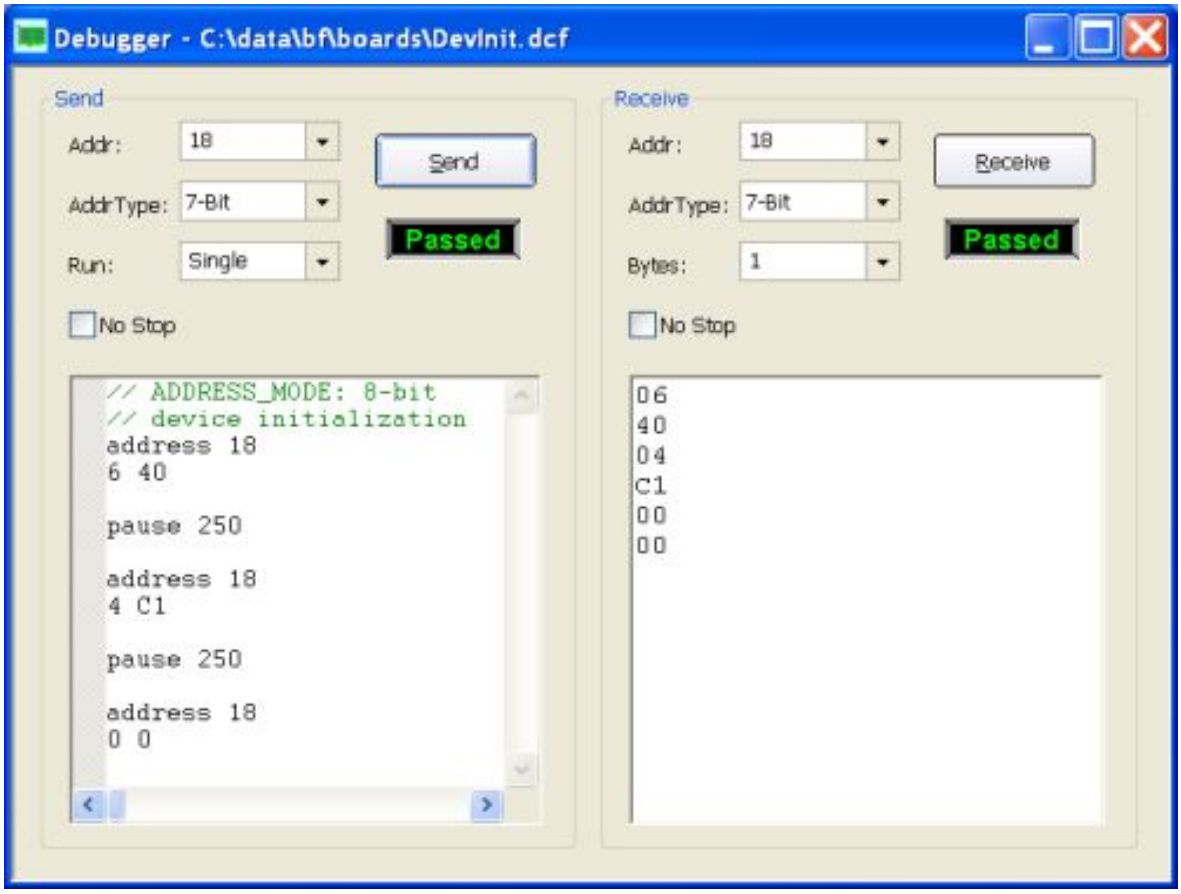


Figure 2.3: Debugger Window.

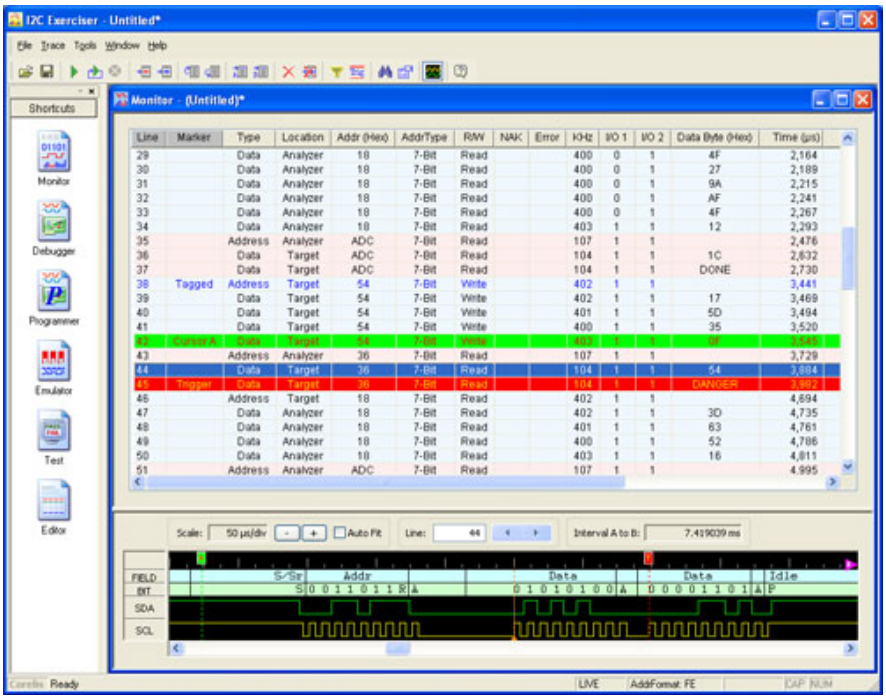


Figure 2.4: Bus monitor window.

Figure 2.5: Scope window

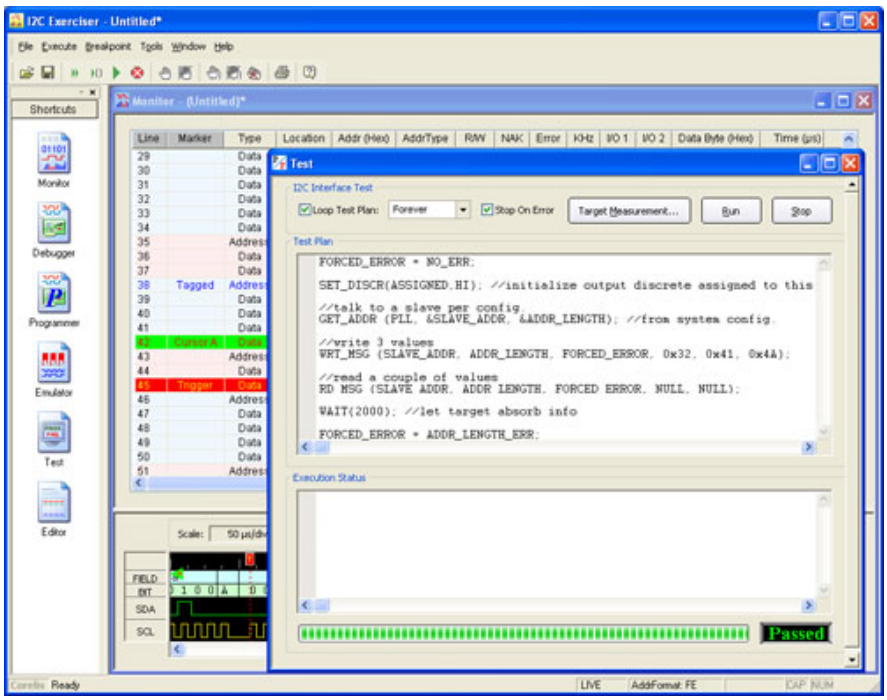


Figure 2.6: Emulator/Scripting language window

2.2 Possible approaches

In this section I will describe the possible approaches to create such an environment for controlling and monitoring digital buses. The software application which should be the main part of the environment ought to be accompanied with a hardware device connected to the computer running the application. This is because the environment should be modular and have the possibility to support very wide range of digital buses including the ones not common on a personal computer. The hardware device should

basically act as a mediator of the analysed/controlled bus to the computer without effecting the computer itself.

Therefore, the whole environment will consist of several main parts. Each of these parts as well as the way these parts will cooperate together offers a series of non trivial problems. These problems and their possible solutions must be deeply analysed before the design and the implementation process. That will be done in the following sections.

The main parts of the environment which will be analysed in this section are the following:

- The hardware device
- The connection between the buses and the device
- The connection between the hardware device and computer
- The PC application
  - User interface
  - Protocol emulation

2.2.1 The Hardware device

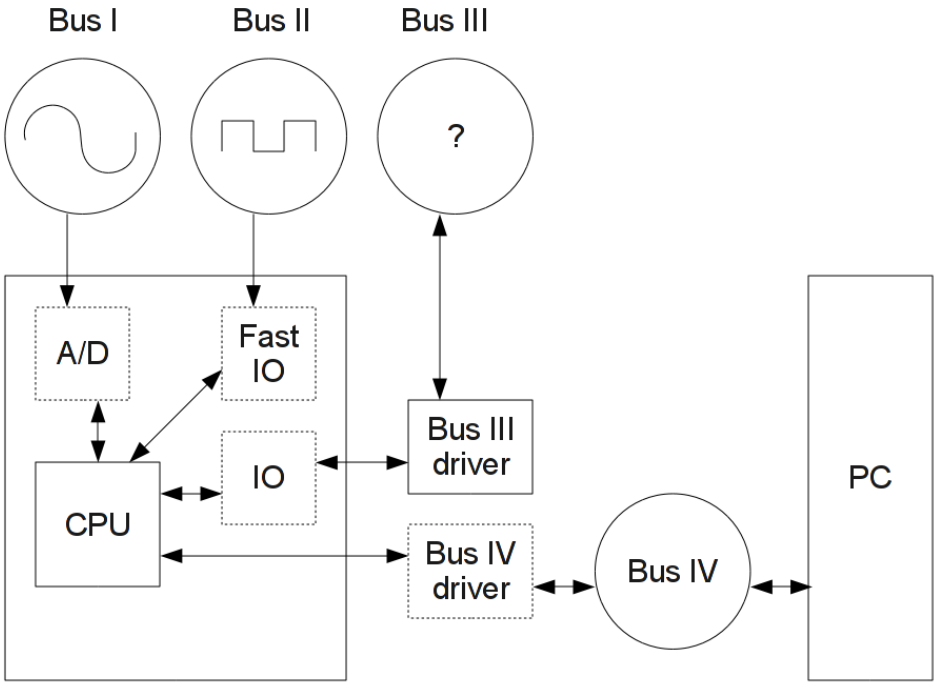


Figure 2.7: The general hardware device diagram.

First lets take a look at what is the devices job and what are the hardware parts of the device which it must have in any case. In general the device will have some main analogue circuit which will process the input signal from the sensing head and pass it on to some analogue to digital converter. This part will be used to analyse the electrical properties of a given bus. Another option is to simply pass the signal to a fast input

pin which will evaluate it either to logical zero or logical one. The second approach should be sufficient for most of the digital buses since most of them don't use more than two voltage levels to differentiate between the meaning of the signal.

Then it must contain an IO circuit to support the desired bus which will be used to gather and send information from/to the bus. And finally it must contain an IO circuit to support connection with a personal computer. Since, typically, there must be at least some digital processing done[1] it will have to have a central processing unit. The figure 2.7 illustrates the aforementioned. To sum up we need these hardware parts at the very least:

- Central Processing Unit
- At least one of the following:
  - Analogue to Digital conversion support
  - Fast input pins
- IO device(s) to support the bus(es) of the interest.
- IO device to provide an interface for connection with a computer.

From the logical point of view, the general work the device does can be summarized in the following simplified steps:

1. Initializes its main peripherals, loads device drivers, initializes the interrupts and gets the device ready to do what its supposed to do.
2. One of these actions:
   - Scans the data from sensing head converts it to digital/logical values and sends it to the computer.
   - Scans the data from analysed bus and sends it to the computer.
   - Receives the data from the computer and sends it to the analysed bus.
3. Stops work or goes back to 2.

Central Processing Unit

Since some processing has to be done on the device itself a CPU is a must. Even though my intention is to do as least of processing on the device as possible it is for certain that it can't be completely avoided. From the main characteristics of the problem a SOC (System on chip) solution is definitely the best choice for me for several reasons:

- SOC chips are available for very fair prices.
- Can be programmed in common environments using standard languages such as C/C++.
- Development boards are easily available.
- It simplifies the electrical design.

[1] For instance to communicate with peripherals or preparation of data for transmission.

- It has most of the peripherals I need built-in.
- It offers good processing power.

There are thousands of different SOC chips from hundreds of different vendors available on the market today. In general, SOC chips can be separated to several different classes though. For simplicity I separated them into three different categories even though this doesn't exactly picture the reality:

- Low End (ie. The ARM7 architecture, The Atmel 8-bit AVR architecture)
  - (-) Offers very limited amount of RAM & ROM, usually tens of kilobytes.
  - (-) Doesn't support 'common' operating systems due to lack of MMU and sufficient amount of memory.
  - (-) Offers less processing power (tens of MHz).
  - (+) The electronics necessary for it to run is fairly simple.
- High End (ie. The ARM9 architecture)
  - (+) Offers high amounts of RAM & ROM, usually tens of megabytes.
  - (+) Has large cache memory (tens of kilobytes).
  - (+) Supports common operating systems such as Linux or Windows.
  - (+) Offers high processing power (hundreds of megahertz)
  - (+) Offers much more advanced peripherals on chip (such as LCD driver, SD card IO, Ethernet IO,...)
  - (-) The electronics necessary for it to run is usually very complex and requires serious knowledge in electronic engineering.
- Extreme High End (ie. The ARM11 architecture)
  - (+) Basically adds specialized upgrades to the High End class.
  - (+) Contains even more advanced peripherals
    - graphics accelerators with OpenGL/ES support
    - floating point units
    - vector arithmetic support
    - ...
  - (-) The electronics necessary for it to run is very complex.
  - (-) Hard to use 'native code' without an Operating system running on board.

2.2.2 The Connection between the buses and the device

The basic idea is that either an external controller IC[2] for a specific bus will be connected to the CPU and will be used to drive the communication on this bus or the bus controller will be embedded into the used micro controller. Usually the external IC controllers provide a standardized way to communicate with them over some common digital bus, most commonly the SPI bus. However, this is not sufficient for the buses

[2]Commonly called bus driver

which use higher transfer speeds than the maximum SPI speed which is up to 70MHz. Since my intention is to support I2C and CAN busses in the first version of my environment then either one of the external SPI driven IC bus controller approach or the bus driver embedded in the micro controller is sufficient enough.

For high speed buses (for instance: Ethernet, PCI, PCI-E, AGP, ...) however, a different approach will have to be considered. Since most of the high end SOC chips provide a system bus (data + address bus) which can handle very high speed transfers it makes sense that this different approach will be using the system bus to connect to the bus controller. For better understanding see the figures 2.8 and 2.9.

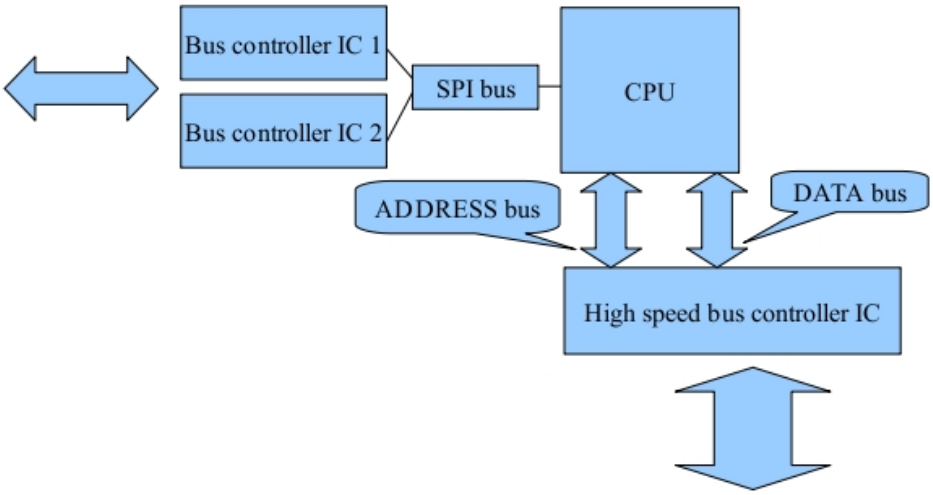


Figure 2.8: Basic diagram of connection between the main CPU and the buses. External bus drivers used.

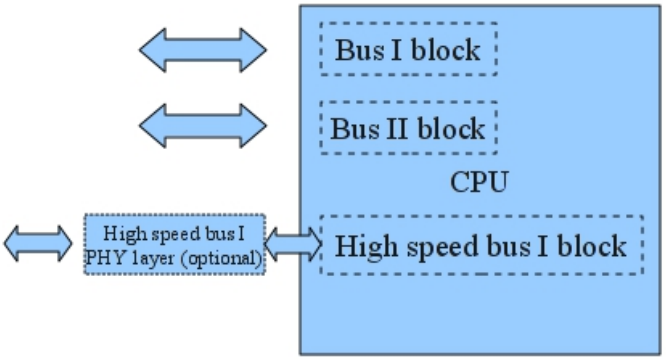


Figure 2.9: Basic diagram of connection between the main CPU and buses. Bus drivers embedded in the main CPU. High speed bus blocks possibly connected to external Physical layer ICs.

For both the CAN and the I2C buses there are many controller ICs available on the market. Most of them also support control over SPI bus and are fairly cheap (ranges from ˜$5 - $25).

2.2.3 The Connection between the device and a computer

Given a common personal computer there is only a limited amount of ways to connect an external device to the computer. In the following text I will list most of the possible connections and analyse their pros and cons:

RS232 serial / COM port

PROs:

- Easy to use
- Lots of code and documentation online

CONs:

- Very slow – only tens of thousands of bauds
- Antiquated (most modern PC motherboards don't include them anymore)

LPT Parallel Port

PROs:

- Easy to use

CONs:

- Very slow and antiquated

PCI / PCI-E bus port

PROs:

- High speed bus
- Peak transfer rate of 133 MB/s
- Good support in common operating systems

CONs:

- Can only be used with a computer offering a PCI/PCI-E slot (no laptop etc.) or appropriate converter.
- Complicated device driver development.
- Quite complex electronics.

USB bus

PROs:

- More or less a standard way to connect most peripheral devices to the computer.
- High speed (ranges from 1.5Mbit/s - 480Mbit/s).
- Usually supported by the high end SOC chips.

CONs:

- Need to write the device drivers (at least the top layer of class drivers)

Ethernet

PROs:

- High speed (ranges from 10Mbit/s – 5Gbit/s).
- Wide operating system support.
- A lot of documentation available.

CONs:

- Big data overhead for direct connections which use small data payloads

From the list above it is quite obvious that the best way to connect the device to the computer is either using the USB bus or the Ethernet. During the analysis of the existing solutions I noticed that most of the products available on the market today support only one way to connect to the computer (most common is the USB). In my solution I would like to support several connections between the device and the computer. For instance: the user should have an option to choose which connection he or she would like to use. In this way if a protocol of communication between the device and the computer is properly defined the user will have an option to create his/her own hardware device which can be then used with the software I will implement during my work.

2.2.4 The main software

User interface

The user interface can basically be either text based or graphical. Both have their pros and cons but in my work the software will be using a GUI mostly due to its complexity.

The biggest problem of a graphical user interface is usually its portability. However, several open source libraries are available on the Internet what solve this issue very well. To name a few of them:

- GTK+
- QT
- wxWidgets

The other problem of a graphical user interface can be its so called user-friendliness. Ideally some "GUI design pattern" should be used to create the layout of the programs parts. Unfortunately, I was unable to find such a design pattern.

During the software design phase (see chapter 7) of the work I decided it would be the best choice to leave the design and implementation of the module specific windows purely up to the developer who implements the given module. It removes the need to update or change the source which the developer doesn't know or doesn't know how it works. In this way only a very simple main window is needed and possibly some support windows for some basic settings and extensions (such as the bus signals visualisation etc.)

Protocol emulation

The protocol or device emulation will be used to emulate a device on the bus being controlled. There are two possible ways how this could be done.

- Using a GUI
  - Manually emulating the device by using common user interface objects.
- Using a scripting language
  - Automatically simulating a virtual device which acts as a real one physically connected to the bus.

I am certain that using a scripting language is the best option for several reasons:

- Well defined syntax
- Possibility of creating libraries of devices
- Easy to transfer

I will use some scripting language which already exists and is well designed for protocol / device emulation and is free of charge. Ideally there should be a possibility to add a support for another common scripting language to the environment.

Chapter 3

The fundamentals and main design decisions

In this chapter I will describe the essential structure of the environment and the choices of specific software and hardware parts of the environment based on the previous analysis. In the beginning the basic definitions of terms used through the following chapters are described.

3.1 Basic definitions

3.1.1 Hardware device

By hardware device it is understood the device which is physically connected to the bus we want to analyse or control and to the users computer.

3.1.2 Transfer bus

By transfer bus it is understood the data channel which is used to transfer data either from the hardware device to the computer or from the computer to the hardware device.

3.1.3 Digital bus

This sections describes what is understood by the term *Digital bus* in this work. A digital bus is a set of wires (or other common medium such as pcb[1] connections etc.) used for communication between two or more devices. This set of lines can be split into two main categories:

- Data lines
- Control / Clock lines

The amount of data lines is given by:

$$DL_{cnt} \equiv 2^n, n \in [0..5] \tag{3.1}$$

If the amount of data lines is 1 ($n = 0$ in 3.1) the bus is called **Serial** and in the other cases it is called **Parallel**.

[1] Printed Circuit Board

The amount of the control lines depends on the principle of the operation of the given bus. In case the bus is controlled only by a clock then the clock line is the only control line and data lines are considered valid if the clock line is in a pre-defined state (ie. up or down). The speed of this clock is called AB_{clk} and represents the speed of the clock in hertz. In case the bus is controlled by several control lines instead, then the meanings of these lines, principle of operation and data validity is purely up to the specification of that bus (including the bit timing specification). However, I assume that there is always at least one clock line or a mechanism which is equivalent with clock (For instance precise bit time / sample time specification as in the CAN bus). The figure 3.1 summarizes the basic bus categories.

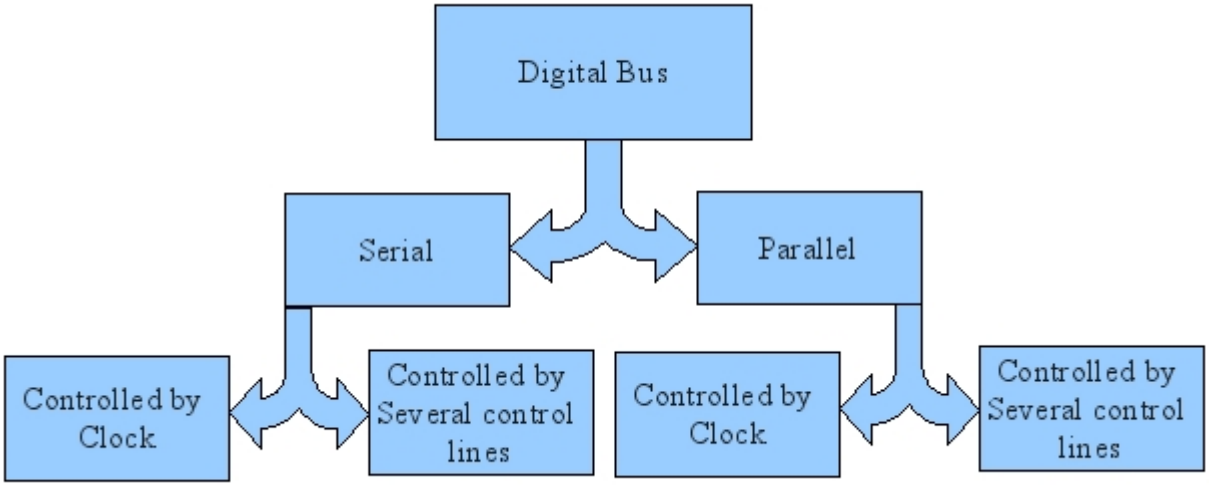


Figure 3.1: Basic bus cathegories

3.2 Main system parts

The whole system consists of several basic parts. They are the following:

- Communication protocol
- Hardware device
- Transfer bus
- Bus analysis
- Bus control

In the following sections these parts are described in more detail, and the most important final decisions (ie. which device was chosen, which transfer bus, etc.) are explained.

3.2.1 The communication protocol

In this section I will describe the main communication protocol which will be used in the communication between the analyser/controller device and a personal computer. It will be helpful during the software design phase of the project. It will be also used by the developers as a first guide on how to design their hardware device.

Overview

The main parts of the environment which the protocol assumes are the following:

- The user is running the software application given with this work on a personal computer.
- The hardware device is connected to the PC via a transfer bus (for instance: Ethernet, usb, PCI, ...)
- The hardware device is connected to a bus (for instance: i2c, CAN, ...) which it is analysing or controlling.
- The firmware of the hardware device is following the protocol specified in this work.

Note that this section specifies the main general rules. For precise specification and implementation details see Chapter 5.

3.2.2 Protocol overview

This section briefly describes the whole protocol so that the reader will better understand the following chapters. The protocol as a whole has basically four main parts. They are the following:

Bus data analysis and preprocessing

This part takes care of the actual data capture from the bus and basic preparation of the data for the transmission.

Bus control

This part takes care of processing commands from the PC and sending them on a given bus in appropriate format. It is assumed that the controlling is physically done via some IC[2] controller determined for the given bus which is externally connected to the main processor or directly embedded inside it. Therefore the possibilities of operation are fully dependant on the bus driver.

Data transmission

This part takes care of the transmission of the data between the device and a computer. It is assumed that the physical transmission is done via some standard high speed bus such as USB or Ethernet. However, the protocol supports adding support via modules for additional buses and encapsulates them in such a way that the data will always 'look the same' on the PC application no matter how they were physically delivered and what other data are needed for the given transfer bus (ie. ARP headers for Ethernet etc.)

Data processing and presentation

This part takes care of the processing of the received data on the PC side and presenting them to the user in logically correct way (in relation with the bus being analysed).

[2]Integrated Circuit

3.2.3 Principle of operation

In this section I will present the basic flowcharts of the principle of operation of the whole environment. It is divided into several parts so that it is easier to follow.

Bus analysis - the basics

The bus analysis basically consists of four main steps. First, the data are captured by the device. Once the data are captured they are preprocessed for transmission. This preprocessing consists of:

- Encapsulating the captured data into lager chunks of data.
- Encapsulating these chunks into packets with other necessary information.

When the packet P_i is ready for transmission it is encapsulated into an appropriate format which follows the bus spy protocol and put on the transfer bus.

Once the data are delivered to the PC application running on the computer the packet P_i is unpacked and passed on to the appropriate module which takes care of the actual data analysis.

Bus analysis - initialization

PC site

The bus analysis begins when the user starts the PC application and selects that he or she wishes to analyse a bus. Then the basic initialization of the application starts. The basic steps of the PC app. initialization are the following:

- Basic OS initializations.
- Selecting & setting up the bus (module) used to transfer the data in between the device and the PC.
- Selecting the bus to analyse and setting the main parameters used during analysis.
- Loading an appropriate analysis module.
- Connecting with the device and sending it the initialization data.
- Waiting for positive acknowledge from the device or detecting that an error occurred.

Figure 3.2 explains this process in more detail.

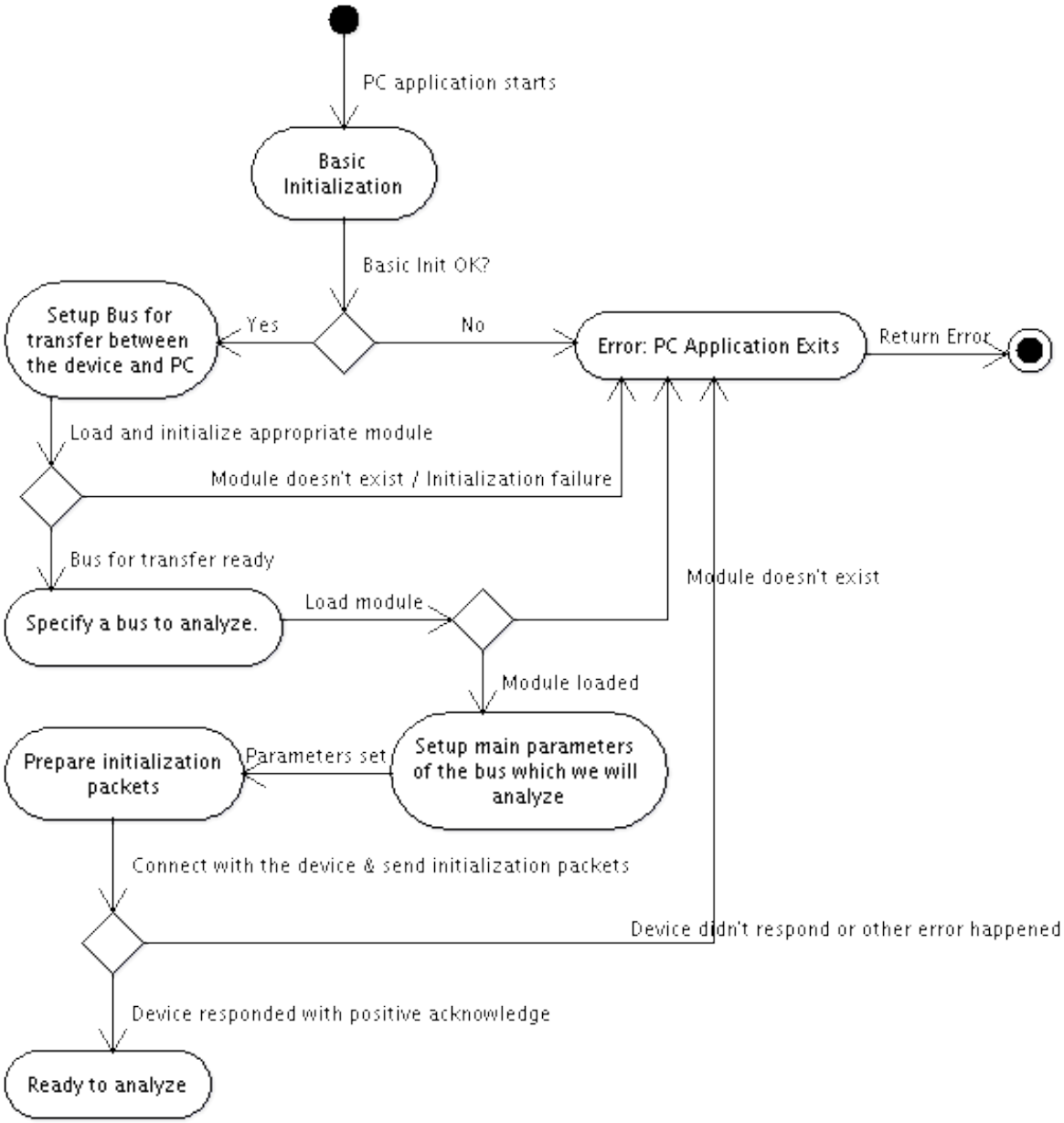


Figure 3.2: Flowchart of the initialization of the analysing environment on the PC site

Device site

On the device site the situation is similar as on the PC site. Once the device is powered on a small initial delay is executed [3]. After that few initialization steps are executed. They are the following:

- Low level system init (IRQ, DMA, buffers, etc.)
- Setting up the bus for transfer.
- Waiting for the initialization packets.
- Once received, following the instructions given in the packets.
- Sending a positive or negative acknowledge.

The Figure 3.3 explains this process in more detail.

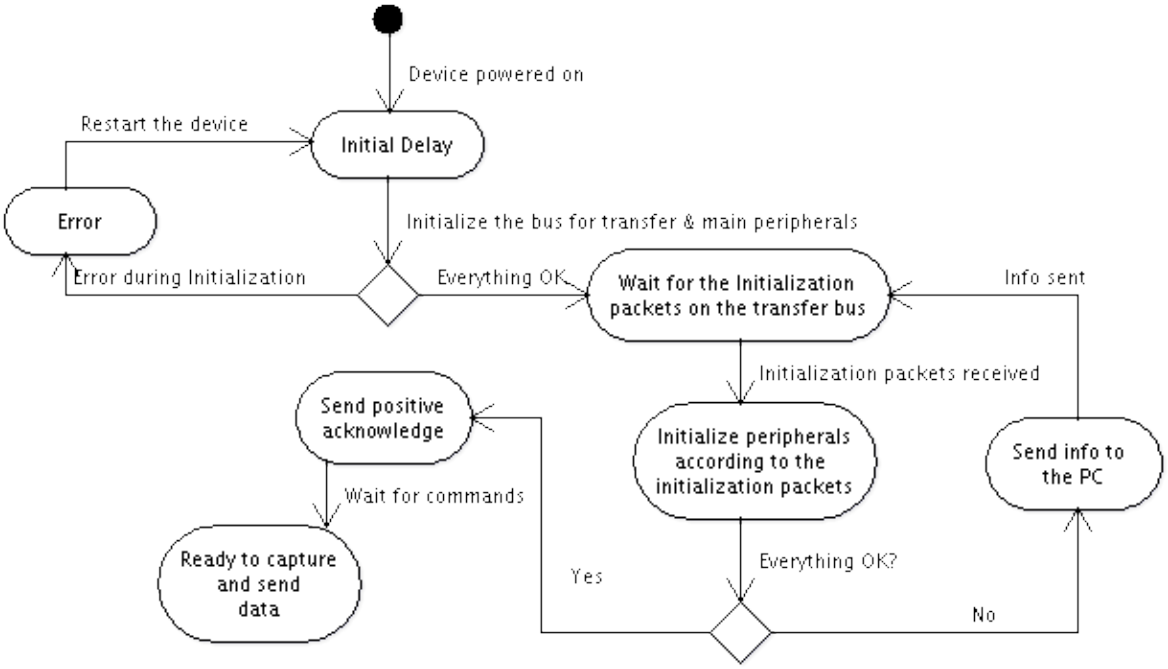


Figure 3.3: Flowchart of the initialization of the analysing environment on the device site

[3]This is a good programming practice. The reason for this is the fact that some peripherals connected to the main processor may or may not need some time to initialize them selves before they can be controlled by the cpu. If they are not given the time to initialize however, the results can be fatal.

Bus analysis - the analysis

Once the environment is initialized as described in the previous section 3.2.3 the actual analysis can start. This section explains the process in more detail.

PC site

Once the application is ready to analyse (receive data) it starts by sending a *START* command to the device.

If the device replies with positive acknowledge the actual analysis starts. The PC application holds until the data from the device are ready. Once the data are received they are unpacked and passed to the module determined to analyse the given bus. The module processes the data and takes care of displaying them.

At this point the analysis can either stop or continue by executing the same steps as in the aforementioned. See figure 3.4 for better understanding.

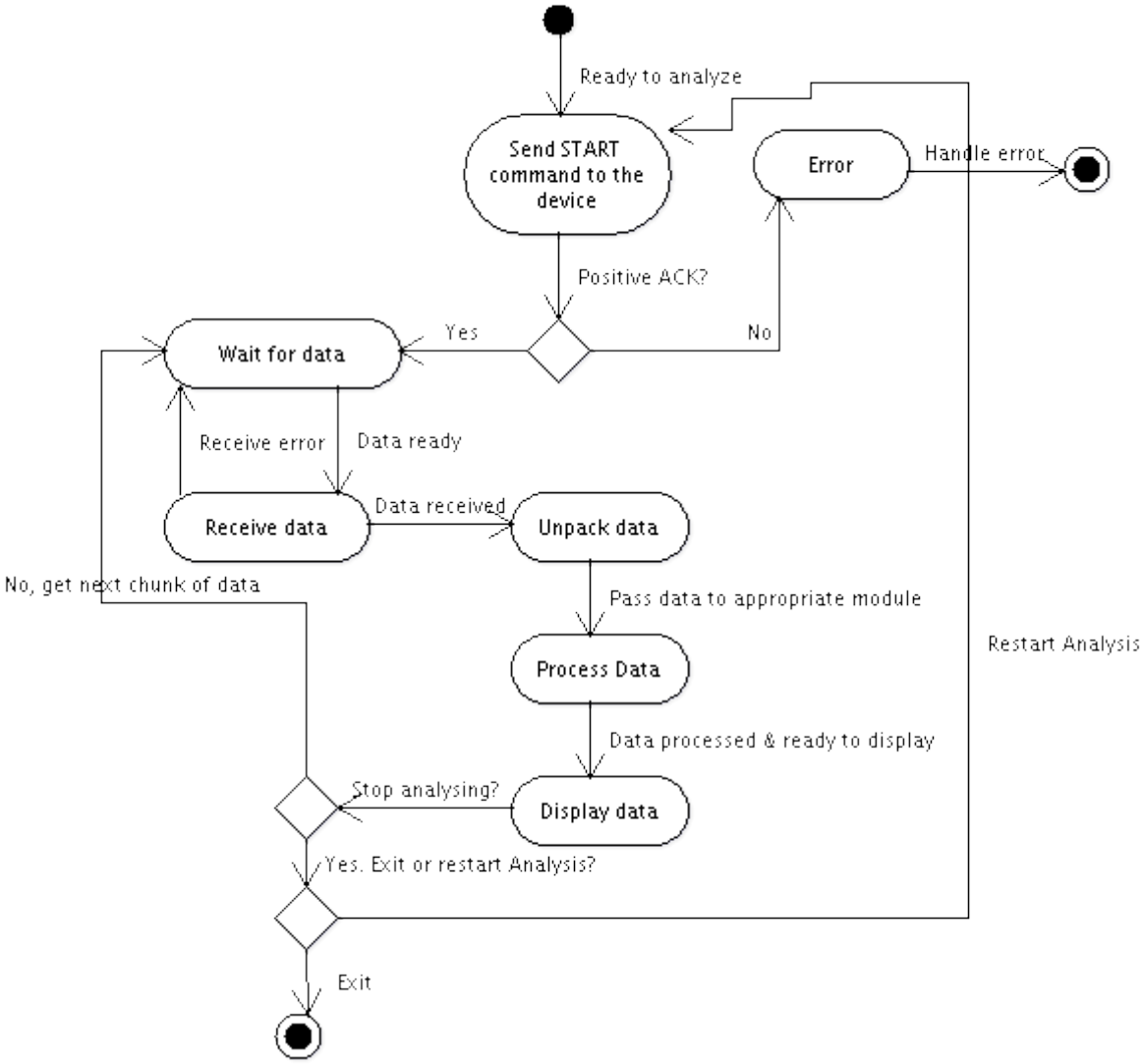


Figure 3.4: Flowchart of the analysis process on the PC site

Device site

On the device site the situation is a bit more complicated for the following reasons:

- We are working in the embedded environment with limited resources.
- We might have to ensure continuous flow of data. In the other case only a limited amount of 'bus time' can be captured (this approach is actually sufficient for a lot of cases).
- The whole process must be fast so that we don't loose any data.

Once the devices receives the *START* command it starts listening on the bus being analysed and when bits of data are available it captures them and stores them. Once the device captures enough bits so that together they shape at least one logically correct unit of data[4] it starts packing the data. The packing consists of:

- Encapsulating the captured bits.
- Adding some extra protocol-oriented information.

Once the packing is done the data are prepared for sending to the PC using a previously defined and initialized transfer bus. If an error occurs during the sending, the device doesn't try to resend the data straight away but rather buffers them and tries to send them later with the newer data. In case the buffer fills completely and there is nowhere to store the data any more the analysis must stop and the device must either let the PC application know that an error occurred or the PC application might detect it by itself. Once the data are either sent or buffered (and buffer is not yet full) the capturing continues again. The figure 3.5 shows this whole process in more detail.

[4]The exact meaning of 'logically correct unit' is purely up to the specification of the bus being analysed. For instance it can be 1 data byte + 1 ack bit etc.

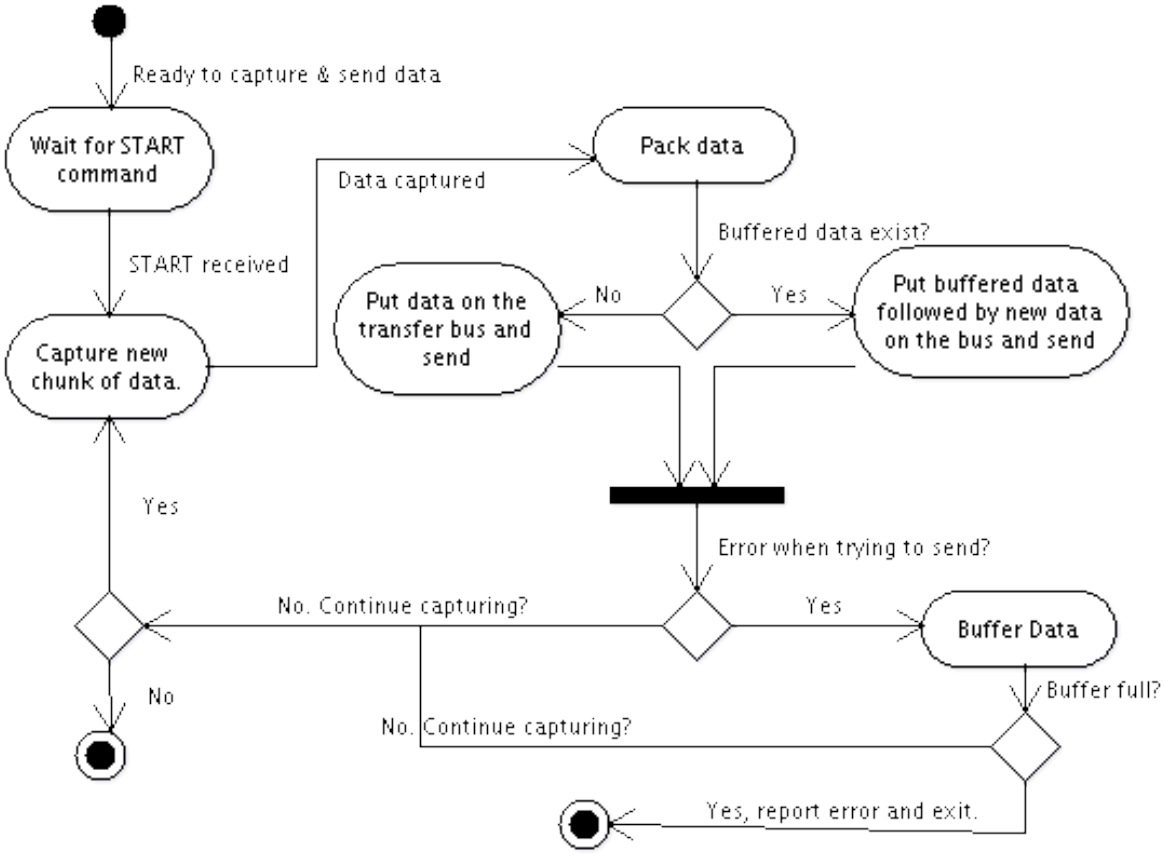


Figure 3.5: Flowchart of the analysis process on the device site

Bus control - the basics

Bus control basically consists of four main steps. First, the device waits for a control command packet. The control command packet consists of a name of an operation and the required parameters for this operation. Once the packet is received these data are unpacked and everything gets ready for the operation to execute (Second step). When everything is ready the command is executed and returned data are packed into a packet (Step three) and sent back to the PC application (Step four).

Bus control - initialization

The initialization for the bus control operation is basically identical to the initialization during the bus analysis operation (Except that control modules are loaded instead of analysis ones of course). To see how the bus analysis is initialized please read the section 3.2.3.

Bus control - the control

As soon as the environment is initialized the control can start. The following sections describe the bus control on a general level.

PC site

When the application is connected with the device and is ready to do the bus control it starts by sending a START command to the device. If the device replies with positive acknowledge the appropriate control module is registered with the control server (which is created once the PC application is started) and the actual control can start.

The control server waits for commands from an external script and passes them to the appropriate control module. Once the control module receives a command to process it wraps all the information to the packet and sends it over the transfer bus to the device. The device always responds in some way, depending on the actual command. The module itself can of course generate commands as well (to be used by the manual control module type). This repeats until the control is finished. The figure 3.6 illustrates this process.

Device site

On the device site the situation during the bus control is very simple from the general point of view. Each device provides a set of control commands (functions) which it can execute during bus control. Once the device is initialized to bus control it waits for a control command, executes it and finally after the command executed the device packs the return data into a packet and sends them back to the computer.

3.2.4 Hardware device

In this section I will list the basic general rules which apply to any device. Then I will list and describe the devices I tried before I made the final decision and describe my experience with them. Lastly, I will briefly describe the device I decided to use. Please note that the detailed description of the main processing board and all other support boards follows in the chapters 4.1 and 4.

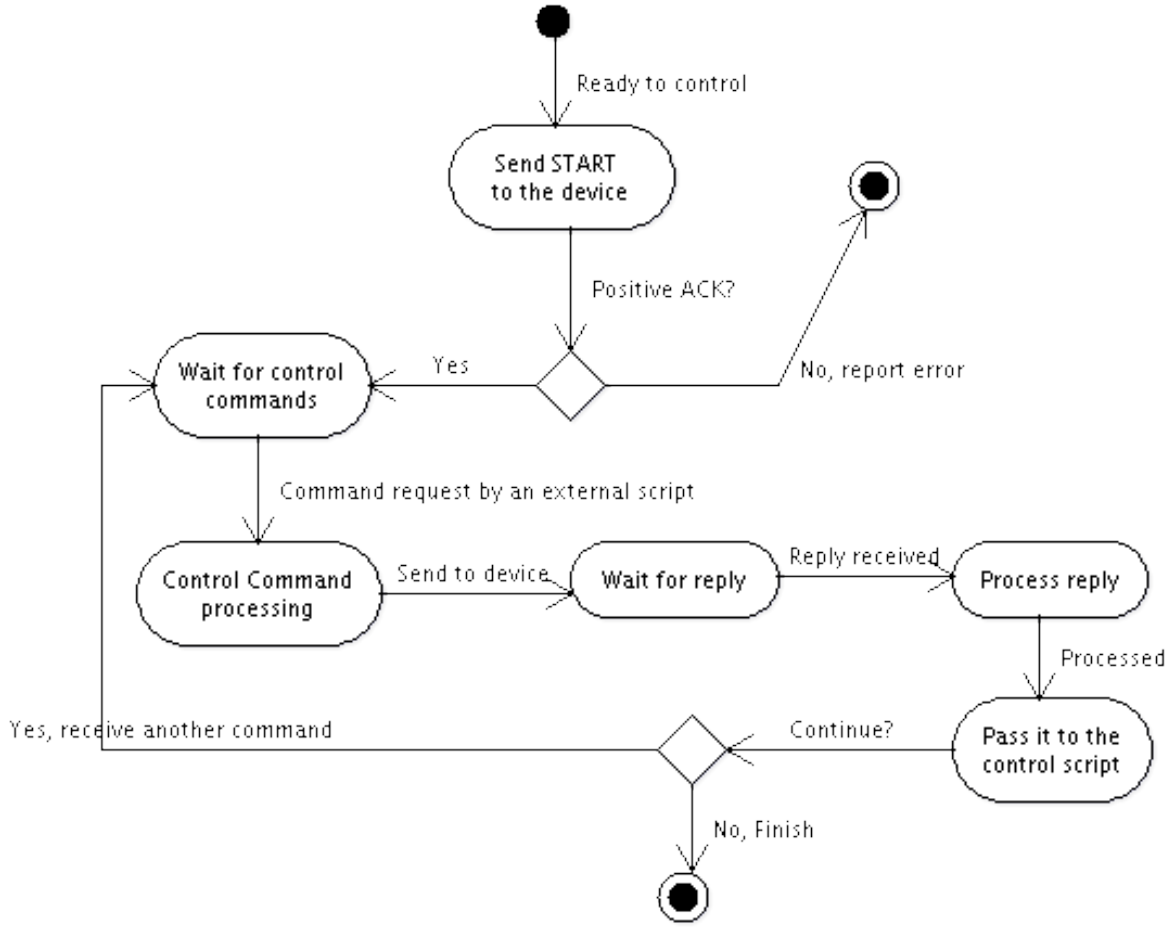


Figure 3.6: The control process from the PC site.

General rules

It is very important to note that the hardware device should be replicable and modular. In other words:

- The user of the software application should be able to create his own device and therefore provide as much processing power as he needs to analyse or control a specific bus. (**replicability**)
- The software in the device implements the protocol specified in this work which allows the device to control or analyse several buses. That way it can ensure compatibility with the existing modules - if it is desired (if not the user can create new module for the same bus which might work in a different way of course). (**modularity**)
- The main processing of the data is done in the software application via separate modules (depending on the bus being analysed). (**modularity**)

Test device: The MDFly development board.

The MDFly development board is based on the very popular LPC210* series chip, the LPCC2103. I have had previous experience with this chip and was very happy with it so I decided to try whether this device would be sufficient for this work. The board contains several peripherals connected to the chip, namely:

Figure 3.7: The MDFly ARM Ethernet development board [27].

- 128x64 pixel LCD color display.
- ENC2810J 10Mbit Ethernet bus controller.
- 2 IO pins
- SD card slot
- 5-way joystick

The chip itself has many embedded features/peripherals in it as well. The most important (from the purpose of this work perspective) follow:

- I2C bus
- SPI bus (unfortunately fully occupied by the LCD)
- Several timers with irq support
- Analog to digital converters

This board/device was the first one which I tried. At first it seemed as it will suffice but during the development several issues came up which more or less made the board impossible to use as an example device for this work. The main reasons were the following:

- The chip contains only 32kB of memory. After the TCP/IP stack was ported to this board however only very few kB of memory were left to use. This was the main issue and it didn't allow continuous analysis.
- There are only 2 IO pins available on the device which showed up not to be enough as I needed some control pins for the peripheral boards.
- The chip does not have an embedded CAN bus block and the SPI bus is unusable as it is 'hard wired' to the colour LCD and the Ethernet bus driver.

Test device: FriendlyARM mini2440 single board computer.

The mini2440 is based on a very powerful S3C24* family chip, the S3C2440. This chip is very powerful and is fully supported by Linux kernel. With some changes in the Linux kernel sources it was working on this platform (the mini2440 platform). I had a lot of experience with this board (including creating my own kernel branch and my own file system with startup scripts) and so I decided to try to use it as an example device for this work. The most important peripherals that this platform contains follow:

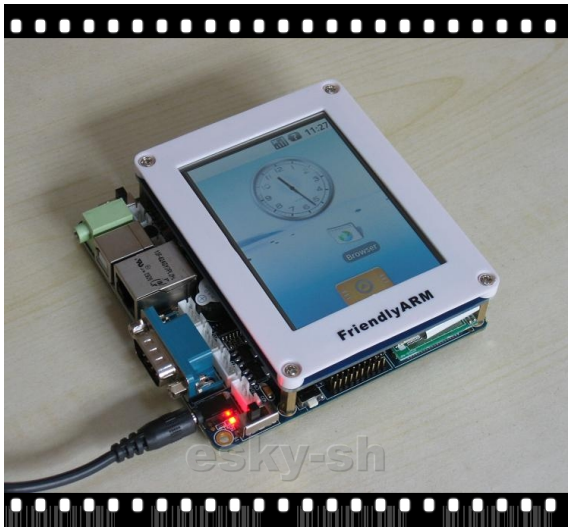


Figure 3.8: The FriendlyARM mini2440 single board computer [28].

- Samsung S3C2440 main CPU clocked to 400MHz
- Ethernet bus driver
- SD card slot
- 64MB RAM
- User buttons
- System bus
- I2C and SPI bus
- Over 10 IO pins

At first this device seemed perfect, however to make it work as an example device for this work would need a lot of Linux kernel and Linux device drivers development which can be extremely time consuming, especially when the lowest level (direct IO pin control) at very high speeds is required. Therefore I decided not to use it and try the final device.

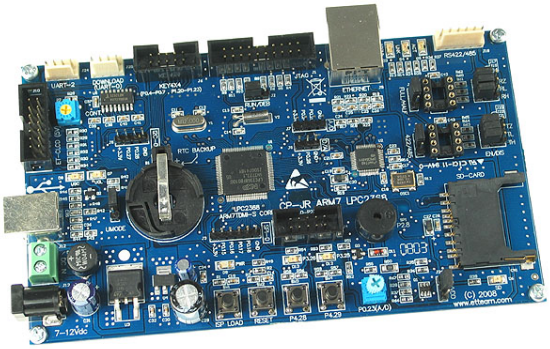


Figure 3.9: The ETT LPC2368 controller board [29].

ETT LPC2368 Controller

Once again this device is based on the very popular Philips LPC series chip, this time a higher class devices, the LPC23** family. The LPC2368 chip is much more powerful than the previously mentioned LPC2103 even though it does have the same core (arm7tdmi). The main reason for this is the peripherals embedded into the chip. The most important follow:

- 512kB of Flash memory
- 58kB of RAM
- USB bus
- 100Mbit Ethernet bus
- 2xI2C, SPI and 2xCAN bus
- DMA for the Ethernet and USB blocks
- SD card support
- More than 10 IO pins available

I decided that this device is perfect for the purposes of this work. It has both I2C and CAN bus drivers embedded into the main chip, high speed Ethernet with the external PHY layer chip on the board and many fast IO pins available. Therefore I decided to use it as the example device for this work. Note that more detailed description of the device and all the support boards follow in chapter 4.

3.2.5 Transfer bus

In this section I will describe the transfer bus which I chose for the connection between the hardware device and a computer. Note that the detailed analysis which describes all possible approaches and their pros and cons is in the section 2.2.3. As an example transfer bus for this work I was deciding between the USB and the Ethernet bus. In the end I decided to use the Ethernet bus. The main reasons for this decision were the following:

- Ethernet is fully supported by all the operating systems which the software given with this work is required to run on (Linux, Windows)
- No need to write device drivers which was not the case with the USB bus
- Similar and very well documented programming interface in the required operating systems
- It is a high speed bus

3.2.6 Bus analysis

In this section I describe what is meant by the term bus analysis in this work and define the basic types of the analysis. Bus analysis means simply viewing all the data which are transferred on the given bus in a non intrusive way (ie. we just passively listen on the bus). Optionally (depending on a specific module and bus type) it can also mean displaying the physical electrical levels/voltages on the given bus. There are two basic types of bus analysis (bus analysing modules). They are the following:

- Zero knowledge analysis
- With knowledge analysis

In the following subsections I define the two types of the analysis.

Zero knowledge analysis

Zero knowledge analysis is the analysis done in the lowest possible level with the least amount of processing done on the actual device. This means that the physical bus signals are scanned in some predefined frequency and the device doesn't try to understand the data but simply packs them and sends them to a computer (in the way which is compatible with the protocol of course). All the decoding of the data, detecting some logically correct pieces and preparing data for visualisation is all done on the computer site by the appropriate module. The bus data sent by the device will typically (this can change for some specific buses of course) look like several streams of zeros and ones where each stream represents one bus line, 1 within the stream represents high voltage level on the bus [5] and 0 represents low voltage. This type of analysis could be, for instance, very useful when analysing the I2C bus. When the device simply acts as a 'digital oscilloscope' errors and incomplete data chunks can be detected by the module running on the computer.

With knowledge analysis

With knowledge analysis basically assumes that the hardware device pre processes the data from the bus in some way (ie gets whole packets/data bytes etc) before its sent to the computer. Therefore the module loaded in the PC application won't do the low level decoding of low level data (voltages on the bus lines) but will be working with logically correct chunks of data. The device doesn't have to contain the bus driver but typically this will be the case. A CAN bus is perfect example where this type of analysis will be very useful. Since errors are detected by every node on the bus it wouldn't be very useful to try to detect them once again in the module. A much easier approach in this case is to let the bus driver detect the error and then send this information to the computer. Also in CAN bus timing is crucial. However to try to meet the timing specifications by simply sensing the voltage level on the bus at a very specific time slices would basically mean implementing own CAN bus driver with listen mode support in the application module. Since many of the CAN bus drivers already implement this feature it would be unnecessary work and waste of time to implement it once more in the software.

In this work both types of analysis modules will be implemented.

3.2.7 Bus control

In this section the types of bus control and what is meant by bus control in this work will be defined. Bus control is an action using which we want to control the bus in some way. Typically this will mean acting as either Master or Slave (Sender or Receiver etc...) on the bus by emulating the specific device in the PC application. For example we might want to talk to some real device connected to the bus or we might want to pretend that some device which understand specific commands is connected to the bus. This can be done in two ways:

- Manual driven control
- Script driven control

[5] Note that in a lot of cases high voltage level actually represents 0 bit

Manual driven control

Manual driven control is the kind of control where the user emulates the device of his choice manually by using standard user control objects (text boxes, combo boxes, buttons etc.). For instance the user might want to emulate reading of data from some specific address as in the standard I2C eeprom devices. In manual way he or she would first set and start the action 'receive n Bytes of data' and once the device actually receives the n Bytes the control module would first display them and then wait for the user to choose the next action (ie. sending some data back in this case).

Script driven control

Script driven control is the kind of control where the device is first fully emulated by some code - scripting language. This code should support all standard programming language features such as conditions, loops, etc. The modules should provide functions which are supported by the given module[6] and these could then be called by the scripting language in loops etc.

3.3 The GUI library

In this section I will present the GUI[7] library which I chose for this work. Note that the analysis of what is mainly required by the library is in the section 2.2.4.

This choice was very important for this work as the software will be quite dependant on the GUI library. I chose the QT[8] library. The main reasons were the following:

- Runs on the 3 major operating systems: Linux, Windows, MAC OS
- Has very well documented C++ interface
- QT provides great tools which ease the UI development
- Has non-free and open source licences
- Lots of literature available

[6] for instance in the C notation: int SendI2CData(int len, char *data)
[7] Graphical user interface
[8] QT is a cross-platform application and UI framework

Chapter 4

System hardware implementation

In this chapter I will describe the hardware device which I designed for the Bus-Spy environment. I named it BS-001 and it is capable of analysing and controlling I2C and CAN buses in their basic form.

4.1 Hardware overview

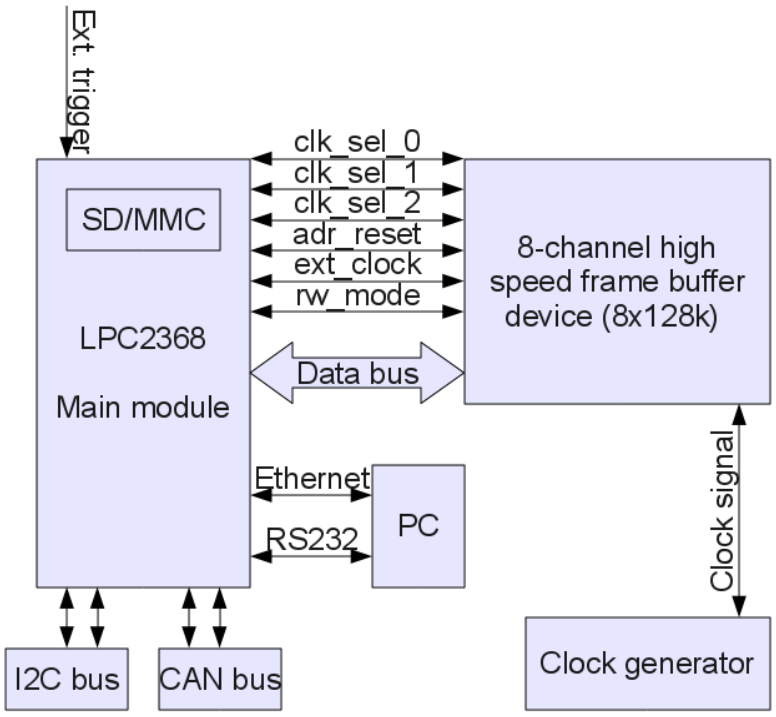


Figure 4.1: Final hardware design overview.

In this section all the hardware devices developed during the development of this work will be briefly described. There are three final devices which were developed. They are the following:

- The BS-001 control and analysis device.

- The i2c eeprom reader device used for testing i2c analysis modules.

- The 2-node CAN led network used for testing the CAN control and analysis.

The BS-001 consists of three main parts (boards): The ETT LPC2368 development board, the high speed 8-channel frame buffer device and the clock generator. The frame buffer device and the clock generator are only used during the i2c analysis at the moment but it (or the principal idea) can be used for analysing basically any bus. The figure 4.1 is a block diagram which illustrates the final complete BS-001 device and the interconnection of its parts. The figure 4.2 is showing the actual hardware including the two test devices. The following sections describe the parts of the device in more detail.

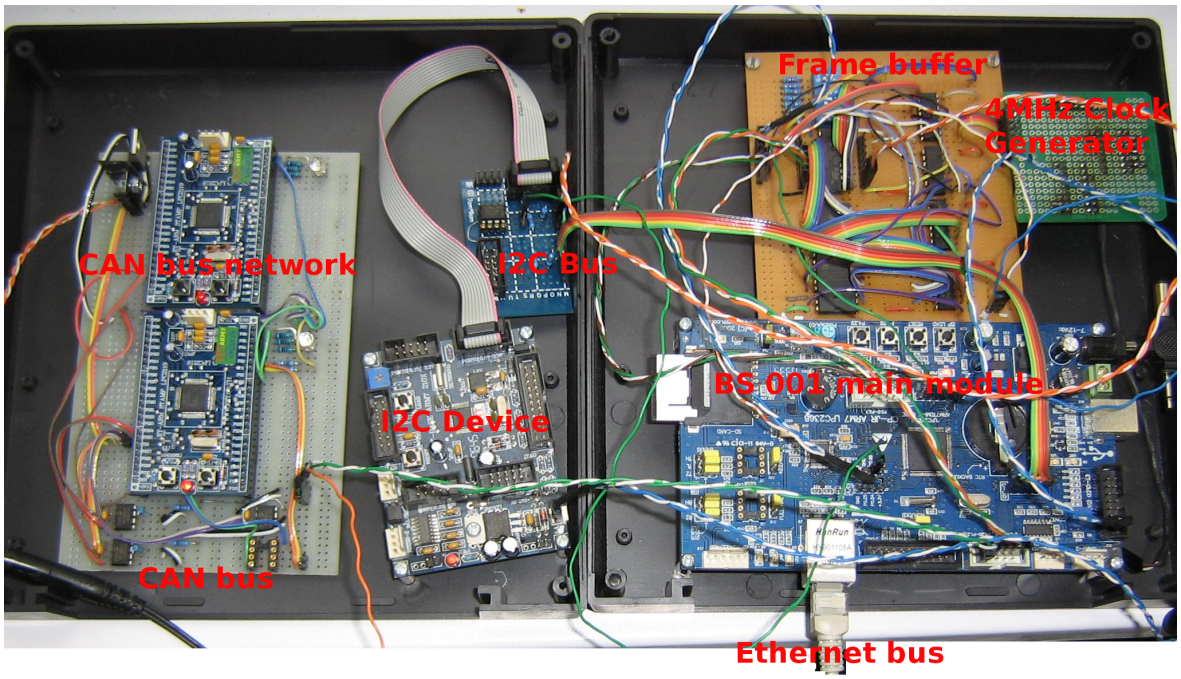


Figure 4.2: The hardware portion of the Bus-Spy environment including two test devices.

4.2 Main module

In this section the main processing module will be described in more detail. As I already mentioned the main processing board is the ETT LPC2368 development board. The layout of this board can be seen on the figure 4.3.

Not all peripherals on the board are used. The ones which are used for purposes of this work are listed in the table 4.2.

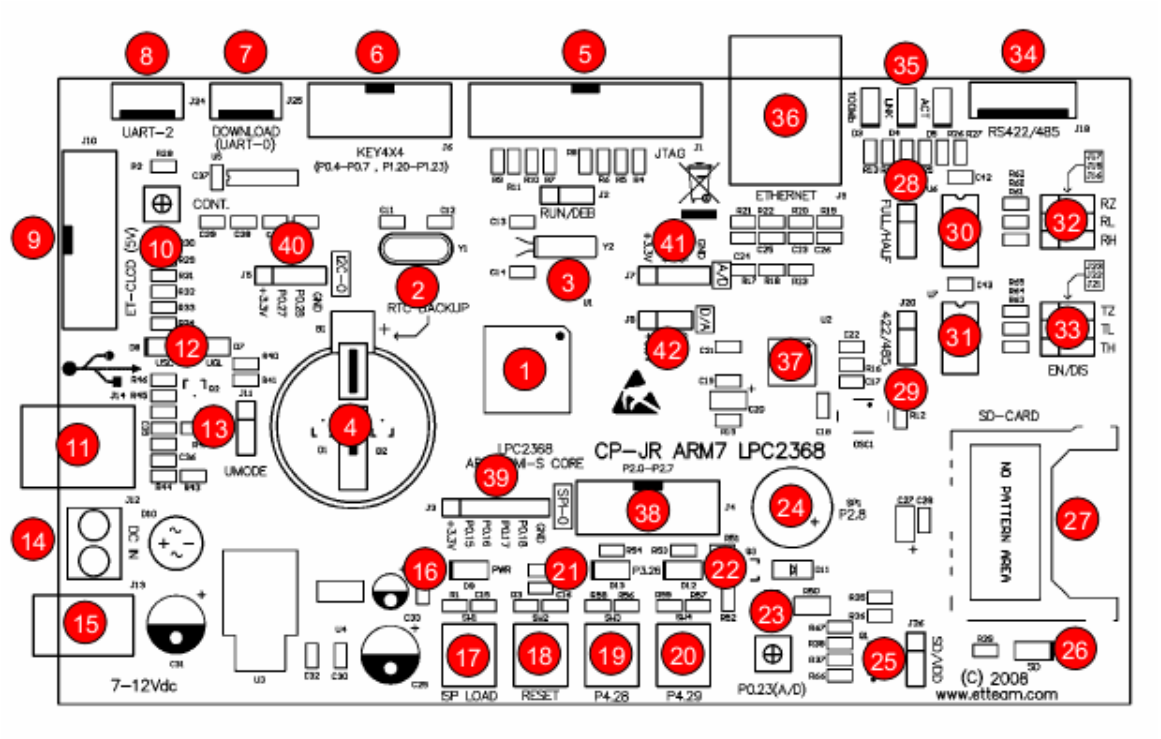


Figure 4.3: The layout of peripherals on the ETT LPC2364 development board. Picture taken from [8].

Num	Peripheral	Usage
7.	RS232-0 channel	Used for debugging prints and diagnostics during development. Also used to interact with the service mode.
17. & 18.	CPU load and reset buttons	Used for loading firmware into the CPUs internal flash.
19.	User button 1	Used to enter service mode during boot.
21. & 22.	Leds 1 and 2	Used to signal the SD/MMC card activity or to inform the user that buffer overflow situation has occured.
35.	Ethernet status led	Used for signalling Ethernet activity.
36.	RJ45	Used for physical connection between the device and a computer (computer network).
37.	Ethernet PHY IC	Used to provide the Ethernet physical layer.
39.	Pins P0[15..17]	Used to select a sampling frequency on the external frame buffer device.
39.	Pin P0[18]	Used for MCU generated clock to the frame buffer device.
41.	Pin P0[24]	Used as R/W mode select pin for the frame buffer device.
41.	Pin P0[25]	Used to reset the address counter on the frame buffer device.
42.	Pin P0[26]	Used as user programmable pin 2.
6.	Pin P1[21]	Used as user programmable pin 1.

6.	Pin P1[23]	Used as user programmable pin 0.
40.	I2C Connector	Used for i2c bus control.
6.	Pins P0[4] & P0[5]	CAN-2 channel of the LPC2368, Used to control and analyse the CAN bus.

Table 4.2: Used peripherals on the ETT LPC2368 development board and their purpose.

4.3 Bus monitoring module

In this section the external frame buffer board will be described. Note that the main idea of the board is based on the logic analyser project from [20]. Also note that the current frame buffer device is a prototype and probably will be upgraded in the future to be able to provide a circular buffer which would in turn allow for continuous real time sampling.

4.3.1 Overview

The frame buffer device consists of two main parts: The clock generating circuit and the sampling circuit. The clock generating circuit is complemented with a 4MHz crystal oscillator which is also the maximum sampling frequency of the device. It is also complemented with a logical divider which is used to divide the frequency to provide more than one sampling frequency options. The sampling circuit provides 8 input channels and is able to store up to 128k samples per channel (131072 samples) and is fully programmable by 6 input pins.

The figure 4.4 shows the 4MHz clock generating circuit. Note that the circuit was taken from the [21]. The parts C2, R1 and R2 determine the cut-off frequency for the clock generating circuit according to the formula:

$$f_c = \frac{1}{2\pi RC}$$

The figure 4.5 shows the frame buffer boards input and output pins. Figure 4.6 shows the address counter and its connection to the external static RAM chip. Finally, the figure 4.7 shows the clock divider (based on a binary ripple counter chip) which divides the clocks into 6 different frequencies. All these frequency signals are then connected to the 8 channel multiplexer which is used to decide which frequency will be used.

On the following pages the schematics of the clock circuit and the frame buffer device are provided. In the next section the principle of operation is described.

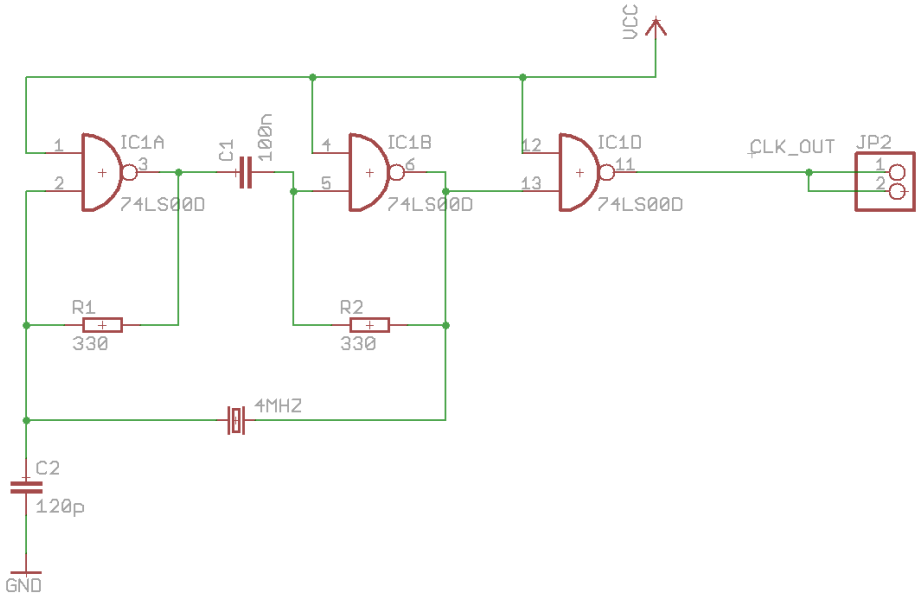


Figure 4.4: The schematics of the clock generating circuit.

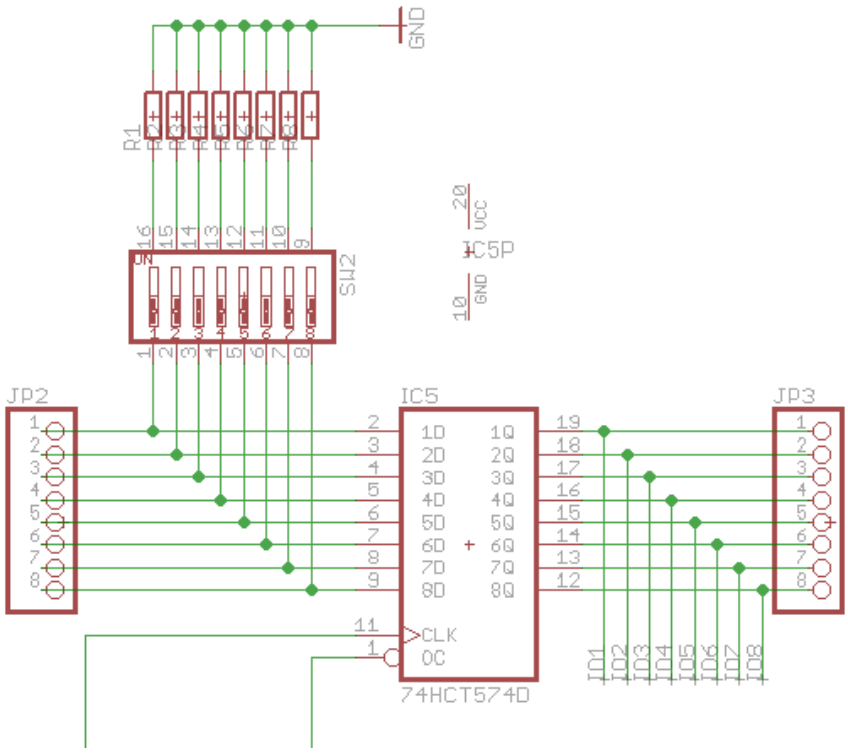


Figure 4.5: The schematics of the frame buffer device 1/3. The 8 input/output channels.

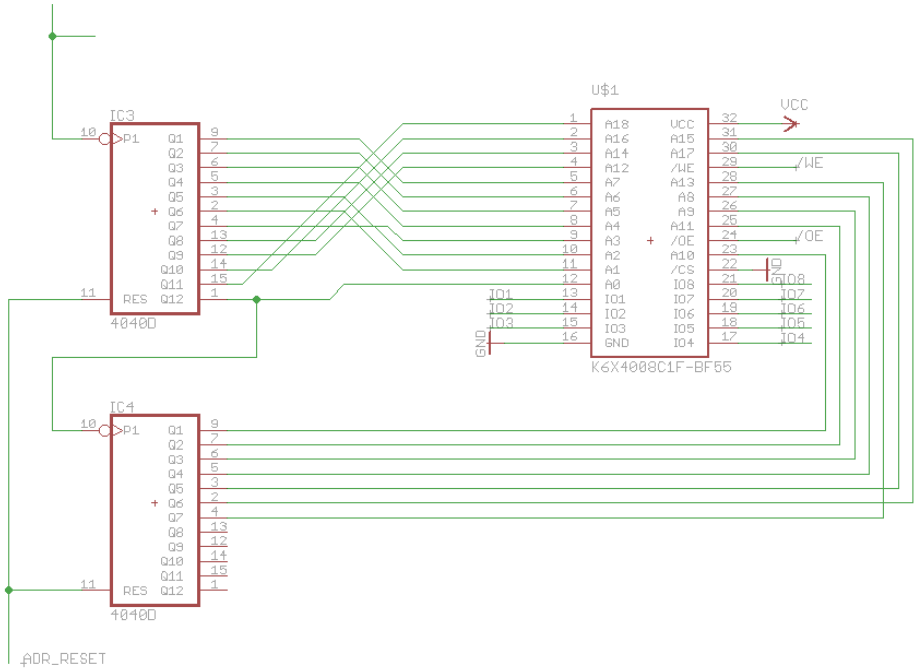


Figure 4.6: The schematics of the frame buffer device 2/3. The address counter and memory.

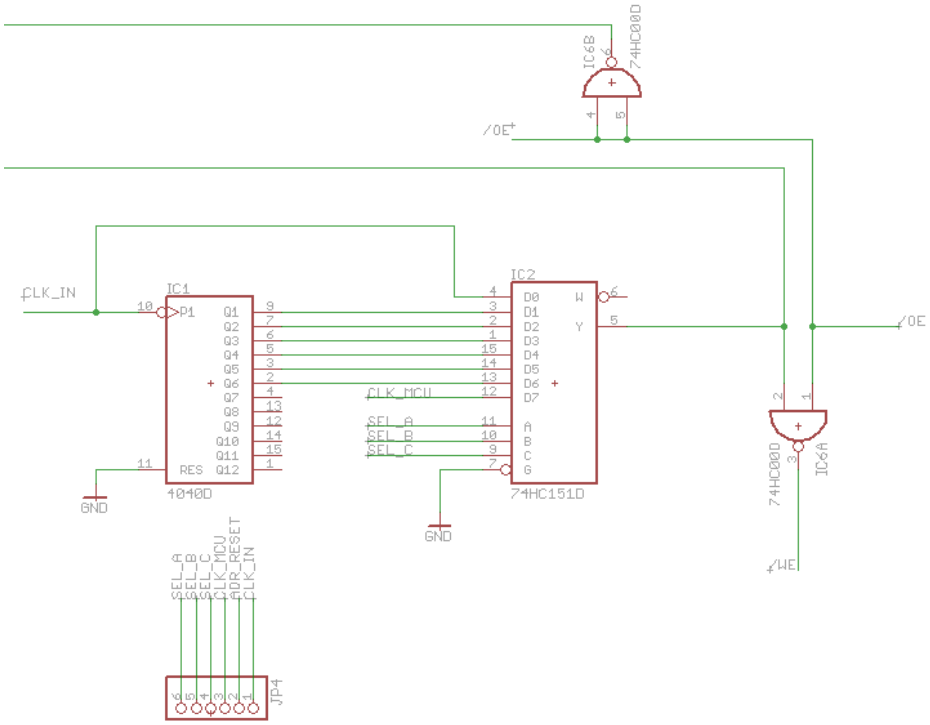


Figure 4.7: The schematics of the frame buffer device 3/3. The clock divider and multiplexer.

4.3.2 Work description

In this section I will describe the principle of operation of the frame buffer device described in the previous section.

First the clock signal is input into the IC1 and D0 input of the IC2. The IC1 4040D binary counter is used to divide the frequency into smaller frequencies and the IC2 74HC151D is used as a multiplexer which is controlled by the main module. The main module chooses the frequency (active input channel of the multiplexer) using the SEL_A, SEL_B and SEL_C inputs as can be seen on the figure 4.7 and the output signal is coming from the pin Y of the IC2. The output signal is then going to one channel of a IC6 which is a logical NAND gate. This gate then controls the R/W pin on the external RAM using the clock signal and the R/W signal from the main module. The clock signal is then divided and drives the IC5 (positive edge trigger circuit) and the IC3 (address counter).

Now lets see what happens in the circuit with every clock pulse:

1. The address is incremented
2. The IC5 flips and puts the values on the input bus to the data bus of the external RAM
3. If writing was enabled (R/W control pin was 1):
    - The gate IC6A produces a logical 0 which enables writing into the external RAM
    - The values on the 8 channels (8 bits) are stored as 1 Byte into the RAM
4. If writing was disabled (R/W control pin was 0):
    - The gate IC6A produces a logical 1 which enables reading from the external RAM
    - The RAM outputs the data at the current address to the data bus which can then be read by the main module

In the end of this section I will describe how the device is operated by the main module using the control pins. There are six control pins all together. They are the following:

- Clock select 0
- Clock select 1
- Clock select 2
- External clock
- Address reset
- R/W mode

The table 4.3.2 lists the options which can be selected by the clock select pins. The external clock pin is used by the MCU to provide its own clock when it is reading the external RAM. The Address reset pin resets the address counter connected to the RAM (IC3 and IC4) if it has positive voltage on it (logical 1). Finally the R/W mode pin is used to choose the operating mode. When it is logical 1 the writing is enabled and reading is enabled when it is logical 0.

SEL_2 SEL_1 SEL_0	Frequency
000	4MHz clock
001	2MHz clock
010	1MHz clock
011	500kHz clock
111	External clock (driven by the main module)

Table 4.3: The clock select options.

4.3.3 Possible upgrades

The main advantage of the described device is the fact that it can be basically used for zero knowledge analysis of any digital bus which consists of up to 8 lines. However, it also has one serious drawback. It can only record time limited amount of bus data because it has only one external RAM. The time which can be recorded can be easily computed using the equation:

$$t_{bus\_time} = \frac{(\frac{RAM_{sizeinbytes}}{8})}{sample\_freq_{Hz}}$$

I believe this drawback could be fixed using an array of larger (512K) external RAMs. This array would then work as a circular buffer. It would have to be electronically assured that once RAM_n is completely filled the next $RAM_{n+1 mod N}$ starts to get filled which could be accomplished by connecting the chip select input pins on the RAMs into some logical circuit similar to the one presented in the previous section. This way the t_{bus_time} from the above equation also becomes the time which is available to read one RAM into some secondary buffer by the controlling device. Of course the circuit would have to allow that one RAM can be read while another one is being filled. This could be acquired by using two separate address counters.

Of course this approach is not completely universal and would require scaling the hardware to ensure some upper limits of the sampling (like what the maximum sampling frequency can be, how many of how big RAMs should be used etc.).

Chapter 5

Software overview

In this section I will describe the software of the whole environment in a very brief way. It will also list the tools used to create this work and the basic hardware and software requirements.

Bus-Spy environment consists of three main parts: The PC software, The hardware device and the embedded software loaded into the hardware device. The environment and the documentation was created using the following tools running on the Gentoo operating system:

- ARM assembler, C, C++, PHP, HTML programming languages
- GNU GCC compiler and utilities [30]
- Gedit text editor [31]
- QT Designer [32]
- Doxygen [33]
- Latex [34]

5.1 Requirements

The minimum requirements to run the Bus-Spy environment are the following:

- Windows (XP or higher) or Linux Operating system
- More than 1.5GHz 2-core CPU
- More than 256MB RAM
- Bus-Spy compatibile hardware device[1]

5.2 Third party software

The third party software used within the Bus-Spy environment is the following:

QT GUI Library [32] Used for the graphical user interface and everything what has something to do with GUI.

[1] Such as the BS-001 introduced in chapter 4

uIP TCP/IP Stack [35] Used inside the BS-001 hardware device[2] to provide TCP/IP stack. Note that this software was ported and changed by me to run on the ARM7 architecture.

lpc23xx EMAC driver Used to control the PHY layer of Ethernet driver.

lpc23xx MCI driver Used to read and write data to the SD card.

PHP interpreter [36] Used for executing external scripts.

5.3 Basic software overview

The PC application is written mostly in C++ programming language. It consists of one main window which is used to select which modules should be loaded and what type of work is about to be done. There are several types of modules and they can cooperate together. Each module can run in a separate thread.

Another important part of the PC application is the control server which is created after the Bus-Spy started. This server waits for commands from an external script and passes them to the appropriate control module in case that this control module is just being used. At the moment only one scripting language is supported - PHP. There are several support classes (ie. class for communicating with the control server) provided so that full control can be done via PHP. However, adding a support for a new scripting language is a matter of two things:

- The scripting language supports shared memory mechanisms
- A support class for communicating with the control server is written

The software running in all the devices (1 hardware device and 3 test devices) is written in ARM7 assembler and C programming languages. Basically the hardware device acts as a state machine which is waiting for events. Once some event happens it is somehow taken care of and either new state is entered or everything starts from the beginning.

5.4 Bus-Spy packet

In this section I will describe the Bus-Spy packet. It is used in the communication protocol (see 3.2.2) between the computer and the hardware device. It is assumed that whichever transfer bus is used for the communication this packet is always sent on the bus as a data payload. It is also assumed that all the communication between the computer and the hardware device is done only via Bus-Spy packets. Therefore the device must understand the structure of the packet. For the PC application modules there are support classes in C++ provided.

The Bus-Spy packet is fairly simple. The structure of it can be seen in the table 5.2. The header consists of four 1 Byte fields (Command, Bus type, Work type, Expect Reply) and four 4 Byte fields (Data size, Custom field 1, Custom field 2, Custom field 3). It is followed by the data payload. The description of all the fields is in the table 5.4.

[2]The device which was provided as the first example device with this work.

Bus-Spy packet			
1B	1B	1B	1B
Command	Bus type	Work type	Expect Reply
Data size			
Custom field 1			
Custom field 2			
Custom field 3			
$DATA_0$			
...			
$DATA_{\frac{Data\_size}{4}-1}$			

Table 5.2: The structure of the Bus-Spy packet. The header is 20B long.

Field	Description
Command	The Bus-Spy command. For instance *START* or *INIT* used during the control and analysis for initialization.
Bus type	The type of the bus we are controlling or analysing. At the moment I2C and CAN buses are supported.
Work type	The type of the work. For instance control or analysis.
Expect Reply	Determines whether reply packet is expected after sending this one.
Data size	The size of the data in the data payload in Bytes.
Custom field 1	32 bits for use by the user. These fields can be used for the control/analysis modules protocols for some extended information, such as initialization parameters etc.
Custom field 2	Same as Custom field 1
Custom field 3	Same as Custom field 1

Table 5.4: The description of the Bus-Spy packet fields.

Chapter 6

Embedded software design

In this chapter I will describe the software running on the BS-001 example device based on the LPC2368 chip. Because the software runs in an embedded environment with very limited resources everything is programmed mostly in the C language. Some hw specific code such as bootstrap or interrupt control functions were written in ARM7 assembly language.

6.1 Modules overview

First I will list and briefly describe all the modules that are used in the software. Note that the files of a module are listed relative to the home directory of the BS-001 project which is located in the *bus-spy/bus-spy-device/lpc2368_ca_device/* directory.

Module	Files	Description
analysis	analysis.c analysis.h	Provides the global variables used for the analysis operations.
armVIC	armVIC.c armVIC.h	Provides support functions to work with the ARM vectored interrupt controller. Note that this module was originally written by R O Software in in-line assembler.
can	can.c can.h	Provides support functions to work with the CAN bus driver. Also provides CAN bus IRQ service routines.
control	control.c control.h	Provides the global variables used for the control operations.
defs	defs.h	Basic definitions used through out the project.
EMAC	EMAC.c EMAC.h	This module was implemented by KEIL - The ARM company. It provides the low level LPC2378 EMAC hardware definitions and a very basic interface to communicate with the EMAC.
fbuffer	fbuffer.c fbuffer.h	Provides basic definitions and operations to work with the external BS-001-FB high speed frame buffer.
i2c	i2c.c i2c.h	Provides basic definitions and operations to work with the I2C bus driver embedded into the LPC2368. Also provides I2C interrupt service routine.
main	main.c	The main module. Controls the operation of the BS-001 device.

std_io	std_io.c std_io.h	Provides some of the standard library stdio.h functions such as printf.
sys_config	sys_config.h	Provides basic definitions used for the low level system configurations.
uip	uip/*.c uip/*.h	The uIP TCP/IP stack ported to the ARM7TDMI-S core.

Table 6.2: The modules used in the software running on BS-001 device.

6.2 Buffering mechanism

In this section the buffering mechanism on the BS-001 device will be described. This mechanism is especially important for the bus analysing work.

There are two levels of buffers. The top level buffer is a 16kB circular buffer split into 512B chunks where one of these chunks is dedicated for the read operations and is not used to store the captured data during analysis. This top level buffer is located in a special location of the LPC2368s RAM which is dedicated for the USB bus block to be used for the DMA. However, because I don't use the USB block I used it for the circular top level buffer.

The second level buffer is a 64MB MMC card. Because the smallest possible block which can be written on a SD/MMC card is 512B it can be thought of as an array of 512B large chunks just as it was defined for the top level buffer. Because speed was a priority I decided not to use any file system on the SD card and just write raw data into the sectors of the SD card. This will probably change in the future: I will probably use FAT file system where there will be only one large file through the whole SD card which will contain the data. This way the analysis could be done in an offline mode on the PC as well since the data could easily be just read from this SD card.

6.2.1 Principle of operation

Once the analysis of some bus has started and some data have been captured everything is firstly stored to the top level buffer array. Whenever there is some free time (for instance: high-speed capture has been finished or we are not servicing an IRQ at the moment) all the buffers are checked and if any of them is *ready* (it was filled with captured data) it is stored to the next free sector on the SD card and marked for transmission. The transfer bus part of the code then detects that some of the SD sectors are ready for transmission, reads them one-by-one into the read chunk of the circular buffer and sends them to the computer with some additional information.

Because both levels use circular buffers it is very simple to detect a data overflow: Once we have data to store and the current buffer is *ready* it is for certain that a buffer overflow situation occurred.

6.3 Initialization

In this section I will describe the initialization process of the BS-001 device. Note that there is also one special kind of initialization - when the service mode has been entered.

The section 6.8 describes this mode in detail.

Once the device is powered on the bootstrap code is entered. This code sets up the stack pointers for each mode of the processor and relocates the .data section from ROM to RAM so it can be accessed faster. After the bootstrap the control is passed to the main() function. In the main function the system is initialized first. That means:

- Setting up the PLL (Phased locked loop). The external 12MHz oscillator is used as a main clock source for the PLL. I set it up so that the processor runs at its maximum speed 72MHz (5x12Mhz).
- Setting up the MAM (Memory acceleration module) for fastest access and the peripheral clock to 72MHz.
- Initializing the vectored interrupt controller.

Once the basic system initialization is finished the system timer is started at 100kHz and the EMAC (Ethernet Media Access Control sub-layer) is initialized. After that uIP is initialized with static IP addresses and set-up to listen on the Bus-Spy TCP port. Finally, the top level buffers are initialized and interrupts are enabled. At this point control or analysis can start.

6.4 I2C Analysis

In this section I will describe the i2c bus analysis from the hardware device perspective. The I2C bus analysis is the *Zero knowledge analysis* as explained in the section 3.2.6. Once the device was initialized and everything was prepared for operation it waits for an Ethernet packet which contains a Bus-Spy packet (see 5.4) as a data payload. If the command field within the Bus-Spy packet contains the *INIT* command the device initializes everything necessary for the requested work according to the initialization parameters sent in the Bus-Spy packet header. In the case of i2c analysis the parameters are the following:

- Data pin
- Clock pin
- Trigger pin
- Sample frequency
- Trigger

Once the device is ready to start the work, it replies with the *INIT_OK* command and starts waiting for the *START* command. When the *START* command arrives the analysis begins.

First the frame buffer device is initialized and the requested sample frequency is set up. Next the device waits for the trigger on the trigger pin and once the trigger event happens the external RAM starts to get filled with the values on the clock and data pins. I use the timer to detect that the RAM was filled according to the following simple formula:

$$t_{sec} = \frac{RAM_{size\_in\_bytes}}{freq_{sample}}$$

When it happens the data are copied from the RAM to the top buffer and to the secondary buffer. When the external RAM is loaded in the top buffer the operation restarts and the device waits for the trigger event again.

Finally, because data were written into the SD card sectors and because i2c analysis flags are turned on the data begin to be loaded from the secondary buffer, packed into a Bus-Spy packet and sent to the computer.

6.5 I2C Control

In this section the i2c bus control is explained from the embedded software perspective. The I2C bus control is the *Script driven control* as explained in the section 3.2.7. When the device was initialized the device waits for the INIT and START commands just like it was described in the previous section 6.4. The main parameter for i2c control is the type of the device which is going to be emulated and the second parameter is decided according to this one. The parameters are these:

- Device type - *Master*
  - Bus frequency
- Device type - *Slave*
  - Slave address

When the *START* command arrives the device starts waiting for the control commands to execute. For i2c control the supported commands and their description is in the table 6.3.

Table 6.3: The control commands supported by i2c control.

Command	Description
WRITE STOP	Sends data to the i2c bus and when done sends STOP condition to the bus.
WRITE RSTART	Sends data to the i2c bus and when done sends REPEATED START condition to the bus. One of the FOLLOW commands is expected after this one.
FOLLOW STOP	This command is used after a REPEATED START was send to the bus. Sends data to the i2c bus and when done sends STOP condition to the bus.
FOLLOW RSTART	This command is used after a REPEATED START was send to the bus. Sends data to the i2c bus and when done sends REPEATED START condition to the bus. One of the FOLLOW commands is expected after this one.
READ	Reads 64Bytes from the i2c receive buffer.
REQUEST CHECK	Checks if data for the slave device are pending or if the i2c slave device should deliver some data.
PICK DATA	Returns data which were delivered to the i2c slave. Should be used only after REQUEST CHECK returned request for picking up data.

PROVIDE DATA	Provides data which the slave should send to the bus. Should be used only after REQUEST CHECK returned request for providing some data.

Whenever the control command arrives the device executes the appropriate function(s) and returns the result in the reply Bus-Spy packet.

6.6 CAN Analysis

The CAN bus analysis is the *With knowledge analysis* as explained in the section 3.2.6. Initially the device waits for the *INIT* and *START* commands again as previously explained in the section 6.4. There is only one initialization parameter for the CAN bus analysis - the bitrate of the bus. When the device receives the *START* command the CAN bus is initialized and appropriate interrupt routines are set up. Next, whenever CAN interrupt occurs the appropriate data[1] are stored to the top level buffer. To prevent the user from waiting for the data due to a low traffic on the bus the time is checked whenever an interrupt occurs and if a predefined period of time has expired even not completely filled buffer is marked as *ready* for transmission. The data sent back from the device are either CAN bus packets or information about a bus error.

6.7 CAN Control

In this section I explain the CAN control from the embedded software point of view. The CAN control is the *Script driven control* as explained in the section 3.2.7. Once initialized the device waits for the *INIT* and *START* commands as in 6.4. The initialization parameters for the CAN bus control are same as for the analysis - the bitrate of the bus. After the *START* command was received the device waits for the control command packets as in the 6.5 section. The commands and their descriptions are in the table 6.5. Whenever the control command arrives the device executes the requested functions and replies using the Bus-Spy packet.

Table 6.5: The control commands supported by can control.

Command	Description
SEND PACKET	Sends a packet to the CAN bus.
ADD LISTEN	Adds a CAN packet ID to be captured by the device if it occurs on the bus.
DEL LISTEN	Removes a CAN packet ID from the queue of IDs we are capturing if they occur on the bus.
PICK LISTEN	Sends all captured packets to the computer.

[1] Typically the data are CAN bus packets but they can be also bus error warnings etc.

6.8 The service mode

The service mode is a special mode entered if the service mode button was pressed during boot. It can be used to change some of the default settings of the device and to do some basic debugging. The functions the service mode provides are the following:

- Print and change IP settings such as IP address or default gateway.
- Initialize SD/MMC card
- Print SD/MMC card information
- Read and dump external RAM on the frame buffer device

Chapter 7

Desktop software design

In this section I describe the design of the main application running on the computer. One of the main requirements was that the application is portable and modular. Both of these requirements were successfully realized. There were some design patterns used during the development. To properly understand them I recommend reading the [2]. The used design patterns are the following:

- Bridge
- Observer
- Singleton

In the following chapters I will first describe the system as a whole and then I will describe the design of the modules of the environment. In the end I will describe the operation of currently finished modules and finally I will describe the principle of operation of the application from when its starts until it finishes. The manual how to add new modules is in the section C.0.6.

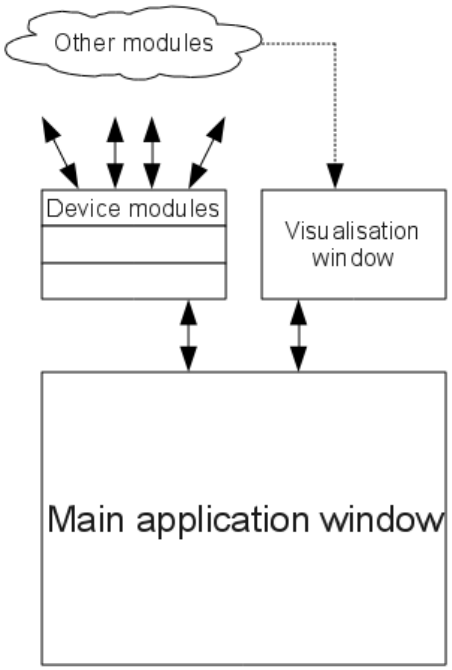


Figure 7.1: The basic overview of the PC application.

7.1 System overview

In this section I will briefly describe the system as a whole. The system basically consists of four main parts as can be seen on the figure 7.1. They are the following:

- The main application window
- The device modules
- The "other" modules

- The universal simple visualisation dialogue

The main application window takes care of everything before the start of some analysis/control operation. All the controls on the window get filled according to the device selection field. The reason for this is the fact that the device module encapsulates all the supported features of a specific hardware device. When some of the basic selections are made by an user several other module-specific windows become available if they are supported by the specific module. The other windows can be the following:

- Device settings window
- Transfer bus settings window
- Control/Analysis module specific settings window.
- Control/Analysis module specific working window.

When the connection between the PC and the device is requested the main window takes care of it using the settings returned by the transfer bus settings window[1] via the transfer bus module. When the main operation starts the control is passed to the control/analysis module specific window until the disconnect action is requested.

7.2 Modules design

In this section I will provide some basic modules design overview and diagrams. The detailed description of each module type is in the following sections.

There are five basic types of modules in the environment. They all share one base class which takes care of registration and organization of the modules. The types of the modules are the following:

- Device modules
- Transfer bus modules
- Controller modules
- Analyser modules
- Server modules

Device modules are used as descriptors of a hardware device. They encapsulate all the functionality the device is able to provide. They provide functions to determine all the functionality and to access all the supported modules.

Transfer bus modules are used to provide an interface to control some transfer bus. They provide the basic read/write operations for a specific transfer bus on a specific operating system.

Controller modules are used to provide support for controlling some bus. They encapsulate their own settings and working windows. They can run in a separate thread.

[1] Only if the transfer bus supports the settings window.

Analyser modules are used to provide support for analysing some bus. They also encapsulate their own settings and working windows. They run in a separate thread.

Server modules are special type of modules. At the moment only one type of server module exists - the control server module. It is used for communication between the control modules and external scripts as explained in the section 7.7.

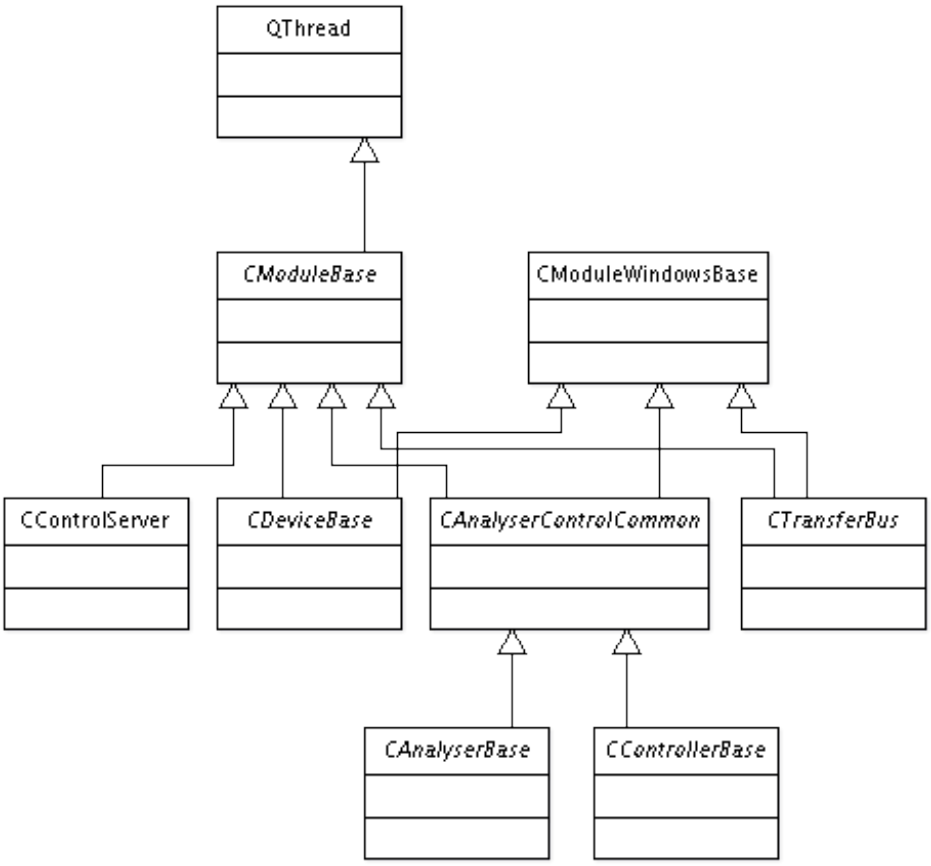


Figure 7.2: The inheritance diagram of the module base classes.

In the end of this section I describe the module class inheritance diagram on the figure 7.2 and the interface of the classes shared between more base classes. Please note that this is not a complete relationship diagram of all classes but only inheritance diagram of the base classes for all of the module types.

All modules base classes share a common predecessor class *CModuleBase*. The *CModuleBase* inherits from QT class QThread to provide portable thread functionality to the modules. The *CModuleWindowsBase* class provides a settings and working window support for the modules. The *CDeviceBase* class serves as a base class for all device modules. The *CControlServer* class is a direct sibling of the *CModuleBase* and acts as the control server module introduced before. The *CTransferBus* class provides the interface of all transfer bus modules. Finally, the base classes for all analyser and controller modules are *CAnalyserBase* and *CControllerBase*. They both share a predecessor *CAnalyserControlCommon* which encapsulates all operations and properties common to all control and analysis modules.

All the modules types are explained in more detail in the following sections. Now I will describe the interface provided by the *CModuleBase* and *CModuleWindowsBase* classes which are common base classes for most modules.

7.2.1 CModuleBase class

The *CModuleBase* class is the base class for all the module types. It provides the basic interface needed by all modules and provides a thread functionality for all modules. The operations it provides and their description follow:

```
CModuleBase(string name,
            string description,
            CModuleBase::module_type_t type);
```
The constructor sets up the name, description and type of the module. It also generates the modules ID and adds it to the module buffer.

```
virtual bool HasSettings() = 0;
```
Determines whether the module has a settings window or not.

```
virtual void ShowSettings() = 0;
```
In case the module has a settings window this functions shows it.

```
virtual bool HasWindow() = 0;
```
Determines whether the module has a working window or not.

```
virtual void ShowWindow()  = 0;
```
In case the module has a working window this functions shows it.

```
virtual void CloseWindow() = 0;
```
In case the module has a working window this functions closes it.

```
virtual bool GetSettingsAsCustomFields(
    custom_field_t (* fields)[NUM_CUSTOM_FIELDS],
    work_type_t work_type,
    bus_type_t bus_type,
    transfer_bus_t transfer_bus_type) = 0;
```
Returns the settings of the module into the fields. The format is compatible with the Bus-Spy packet (see 5.4) custom fields.

```
static size_t GetInstancesSize();
```
Returns the number of module instances which have been created.

```
static CModuleBase * GetInstance(size_t idx);
```
Returns the instance with the ID given in the parameter.

```
string GetName();
```
Returns the name of the module.

```
string GetDesc();
```
Return the description of the module.

```
size_t GetID();
```
Returns the ID of the module.

```
CModuleBase::module_type_t GetType();
```
Returns the type of the module.

7.2.2 CModuleWindowsBase class

This class serves as settings and working windows support for the modules. The settings window is opened when the specific module was selected and the settings button was clicked on the main window near the modules name. The working windows are only used by the control and analysis modules. They are opened once the work was started by clicking on the start button on the main window.

The class provides and implements the following operations to the modules:

```
QDialog * GetWindow();
```
Returns a pointer to the working window dialogue.

```
QDialog * GetSettingsWindow();
```
Returns a pointer to the settings window dialogue.

```
void SetHasWindow(bool value);
```
Sets whether the module has working window or not. Note that working windows are only used by the control and analysis modules.

```
void SetHasSettings(bool value);
```
Sets whether the module has settings window or not.

```
void SetWindow(QDialog *window);
```
Sets the working window to the window given in the parameter.

```
void SetSettingsWindow(QDialog *settings_win);
```
Sets the settings window to the window given in the parameter.

7.3 Transfer bus modules

In this section I will describe the structure of the transfer bus modules in the Bus-Spy environment. The transfer bus modules are used to provide portable support for transferring data between a personal computer and the hardware device.

The design of the transfer bus modules is based on the Bridge design pattern. This design pattern is meant to *"decouple an abstraction from its implementation so that the two can vary independently"*[2]. It is perfect for the transfer bus module design because it will provide basic interface for all the transfer bus modules with the possibility to have different implementations for different operating systems. At the moment only Ethernet transfer bus is supported on Windows and Linux but now it should be quite easy to add a support for new transfer buses.

On the figure 7.3 is the relationship diagram of the classes creating the transfer bus modules. Because transfer bus is a Bus-Spy module with will typically have a settings window the abstraction class *CTransferBus* inherits from the *CModuleBase* and the *CModuleWindowsBase* classes. The *CTransferBus* class also has a relationship with the *CTransferBusImpl* class which defines the interface for the implementation classes. This interface is the same as for the *CTransferBus* class. Finally, the class *CEthernetBus*

is the refined abstraction of the *CTransferBus* class and is used for communication between the hardware device and a computer using the Ethernet bus. The classes *CLinuxEthernetBusImpl* and *CWinEthernetBusImpl* are concrete implementations of the Ethernet bus for Linux and Windows operating systems.

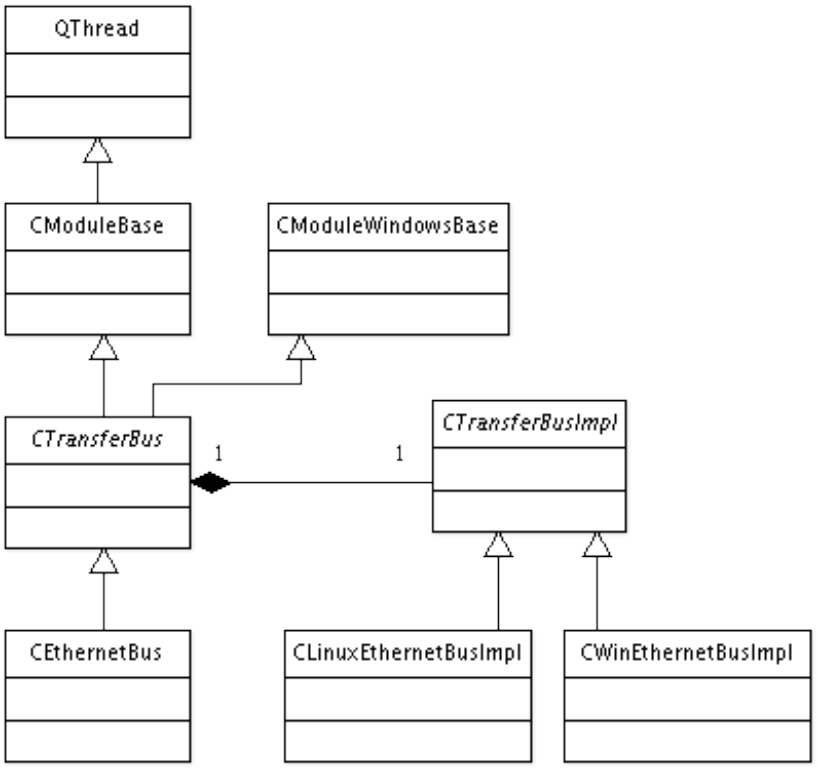


Figure 7.3: The relationship diagram of the transfer bus module classes.

The interface provided by the *CTransferBus*[2] and its description is the following:

```
virtual int DataSend(CBusSpyPacket &data, const CDeviceAddress &adr) = 0;
```
Opens a connection with the device with address adr, sends the data and closes the connection.

```
virtual int DataRecv(CBusSpyPacket &dataOut, const CDeviceAddress &adr) = 0;
```
Opens a connection with the device with address adr, receives data if available and closes the connection.

```
virtual int StreamConnect(const CDeviceAddress &adr) = 0;
```
Opens a stream with the device with address adr.

```
virtual int StreamDisconnect() = 0;
```
Disconnects the previously connected stream.

```
virtual int StreamSend(CBusSpyPacket &data) = 0;
```
Sends data to the previously opened stream.

[2] *CTransferBusImpl* respectively.

```
virtual int StreamRecv(CBusSpyPacket &dataOut) = 0;
```
Receives data from the previously opened stream.

```
bool StreamIsConnected(void) const;
```
Checks whether some stream is connected or not.

In the end of this section I provide a snippet of code which demonstrates how the transfer bus modules can be used and also the strength of the bridge design pattern.

```
// create the implementation
#ifdef __linux__
    pTransferBusImpl = new CLinuxEthernetBusImpl();
#endif
#ifdef _WIN32
    pTransferBusImpl = new CWinEthernetBusImpl();
#endif
// create the transfer bus with implementation created above
CTransferBus *pTransferBus = new CEthernetBus(*pTransferBusImpl);
// we use ethernet transfer bus - therefore ethernet address
CEthernetDeviceAddress adr;
// ... fill in the address details
// connect with device as stream
pTransferBus->StreamConnect( adr );
// create the bus-spy packet
CBusSpyPacket cmdpacket(CMD_INIT, BUS_TYPE_I2C, WORK_TYPE_ANALYSIS, 1);
// send some data
pTransferBus->StreamSend( cmdpacket );
// receive some data
pTransferBus->StreamRecv( cmdpacket );
```

7.4 Device modules

In this section I will describe the structure of the device modules in the Bus-Spy environment. The device modules are used to encapsulate all the functionality that the hardware device supports. The concrete instances of analyser, controller or transfer bus modules are private members of the device modules instances and are accessed purely through the instance of a device module.

The figure 7.4 presents the relationship between the device module classes. Because the device might want to provide a settings window the *CDeviceBase* base class inherits from the *CModuleWindowBase* class. The class *CGZeCADevice* is the only device module currently available in the Bus-Spy environment. It serves as a descriptor of the BS-001 hardware device and encapsulates all its functionality.

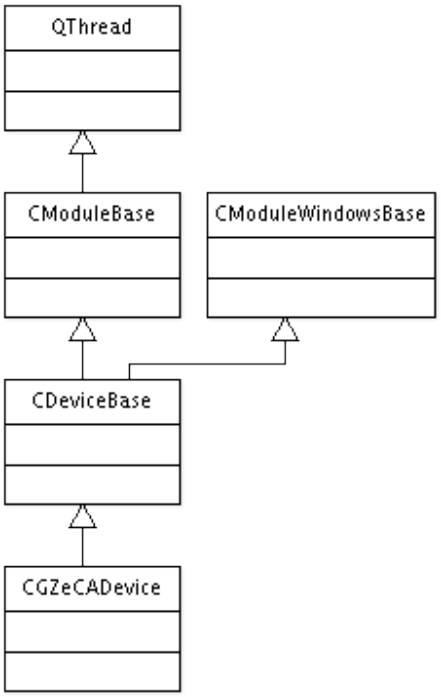


Figure 7.4: The relationship diagram of the device module classes.

The interface the device module base class provides is the following:

```
virtual bool InitTransferBus(transfer_bus_t type) = 0;
```
Initializes the transfer bus type and prepares is for use by this device.

```
virtual bool TransferBusSupported(transfer_bus_t type) = 0;
```
Returns true if the given transfer bus type is supported by the device.

```
virtual bool WorkOnBusSupported(work_type_t work, bus_type_t bus) = 0;
```
Returns true if the work is supported on the bus by the device.

```
virtual bool WorkSupported(work_type_t work_type) = 0;
```
Returns true if the work_type is supported by the device.

```
virtual bool Connect() = 0;
```
Connects the device with the computer using the previously inited transfer bus.

```
virtual CTransferBus * GetTransferBus() = 0;
```
Returns a pointer to the transfer bus module.

```
virtual CAnalyserBase * GetAnalyser(bus_type_t bus_type) = 0;
```
Returns a pointer to the analyser module for the bus_type given in the parameter.

```
virtual CControllerBase * GetController(bus_type_t bus_type) = 0;
```
Returns a pointer to the controller module for the bus_type given in the parameter.

```
unsigned int GetNumInputPins();
```
Returns the number of available input pins on the device which can be used for zero

knowledge analysis.

7.5 Analysis modules

In this section I will explain the structure of the analysis modules classes. The analysis modules are used to implement either the zero knowledge or with knowledge analysis of some specific bus. There can be more modules for analysing one bus of course. It is purely up to the device descriptor class (a sibling of *CDeviceBase*) which module will be used during the analysis of that bus.

The analysis modules also allow a layered structure of analysing modules. That means that one module can use another one to help him implement a top layer of some specific protocol. Hopefully an example will clear this up:

There is one 'with knoledge' analyser module ready for the CAN bus in the BusSpy environment. Now imagine that someone would like to implement a module for analysing a CANOpen protocol which is just an extension of the CAN protocol (to provide upper layers of the protocol only CAN packets are used). He or she can simply use the finished CAN analysing module to acquire all the CAN packets from the bus and use them for the analysis of the CANOpen protocol.

This functionality is accomplished via the Observer design pattern which is a subset of the publish/subscribe design pattern. The upper layer analysers can register them selves as observers of the lower layer analysers which in return notify the upper layer analysers when some logically correct event happens on the bus (ie. full CAN packet was received etc.) and pass the data to them.

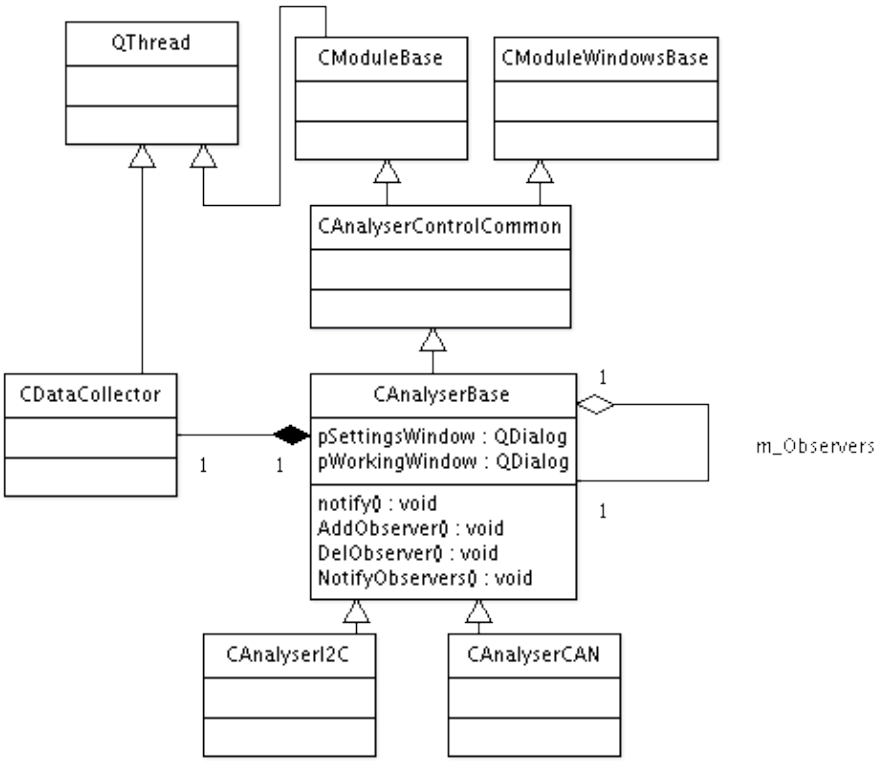


Figure 7.5: The relationship diagram of the analysis module classes.

The relationship diagram of all the analysis module classes is on the figure 7.5. Because its a module which will typically have both settings and working windows

the base class *CAnalyserBase* inherits from the *CAnalyserControlCommon* which in turn inherits from the *CModuleBase* and the *CModuleWindowBase* classes. It also has a relationship with the *CDataCollector* class. The *CDataCollector* class is used to provide Bus-Spy packets from the transfer bus to the module. It runs in a separate thread so it doesn't slow down the work of the analysis module.

The *CAnalyserBase* class also provides the Observer design pattern interface. Note that both observers and notifiers are instances of *CAnalyserBase* in this case. Finally, the *CAnalyserI2C* and the *CAnalyserCAN* classes are the concrete analysers for the I2C and CAN buses. The I2C analyser is the zero knowledge type and the CAN analyser is the with knowledge type of analyser. Both of the analyser modules are used by the BS-001 device descriptor to analyse the I2C and CAN buses.

Once the concrete analysis module has been started it takes full control and uses it own working window to interact with the user. Therefore the module-specific GUI design if purely up to the implementer of the module.

7.6 Control modules

In this section the structure of the control modules classes will be described. The control modules are used to implement the script driven or manual control of some specific bus. Just like with the analysis modules there can be more modules to control one bus and it is up to the device module which control module will be used.

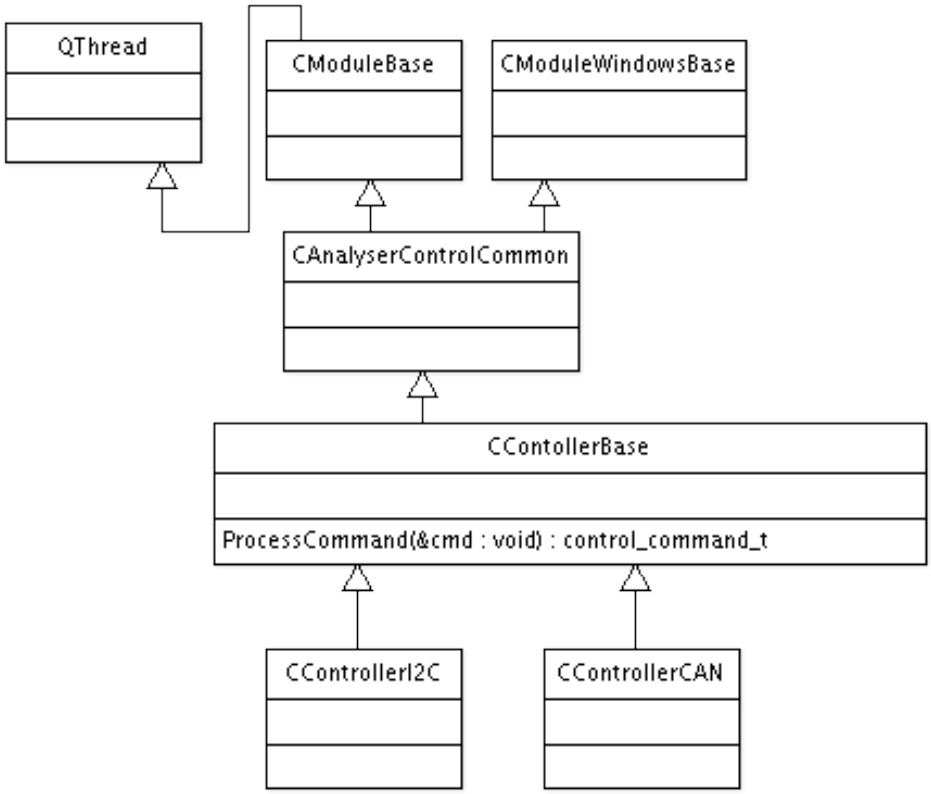


Figure 7.6: The relationship diagram of the control module classes.

The figure 7.6 contains the relationship diagram of all the control module classes. The control module base class *CControllerBase* inherits from the *CAnalyserControl-Common* class which provides the module and module windows support through inheriting from the *CModuleBase* and the *CModuleWindowBase* classes. The classes

CControllerI2C and *CControllerCAN* are the concrete implementations of the control modules. The *CControllerI2C* is the script driven controller of I2C bus and the *CControllerCAN* is the script driven controller of CAN bus. Both these modules are supported by the BS-001 device.

The script driven control is managed by the control server described in the next section. Whenever some action is needed by the external script the control server calls the ProcessCommand method of the controller and passes the requested command as a parameter. The module in turn executes this action and sends the server a reply command which is then passed to the control script.

The manual control is achieved purely by the concrete module itself. This is because only the module knows what operations does its working window support. However, typically the working window will provide some GUI objects using which the user can 'construct the command' and once he or she requests the execution of it the working window would simply call the ProcessCommand method of the controller with some appropriate parameter.

7.7 Control server & external script support

In this section I will describe the control server and how the external script support was achieved. The control server is created when the application was started. The control server is implemented using the singleton design pattern and therefore exactly one control server lives in the environment at any time. Also, exactly one control module can be associated with the server at any time (there can be just one client).

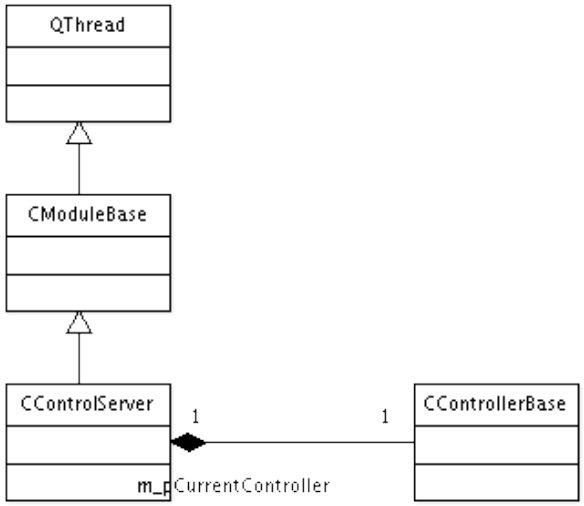


Figure 7.7: The relationship diagram of the control server module classes.

Once the control server is created it requests 1MB of shared memory from the operating system and uses this block of memory to communicate with the clients - the external scripts. This approach was chosen for several reasons:

- Any scripting language which supports shared memory mechanisms can be used. For instance: PHP, Python, Perl
- It removed the need to write my own scripting language which would be a very time consuming task.

When the control server creates the shared memory segment it provides the user with the keys to the shared memory segment and to the semaphore which is used to protect this segment. The protocol between the server and scripts is designed in a way that it can be used without the semaphore (even though its not recommended).

This is because of problematic semaphore implementation in some common scripting languages. The protocol rules are the following:

- After start the server waits until some controller is assigned with it.
- After controller was assigned to the server it waits for a "script command header" which occurs in the shared memory.
- The script must ensure that the command header is the last thing which gets written to the shared memory segment (no semaphore protection).
- When server detects the "script command header" it loads all the data from the shared memory segment, encapsulates the data into a format compatible with the control modules and passes it to the control module which is currently associated with it.
- The control module executes the requested command and always returns a reply command to the server no matter how the operation went.
- The server unpacks the data it received from the control module and stores it in the shared memory segment in an appropriate format.
- When the server stores the reply to the shared segment it waits for the "script command header" again.

The structure of the classes related to the control server module is on the figure 7.7. It is very simple: Because its a module without windows the *CControlServer* class inherits only from the *CModuleBase* class. Other than that, the *CControlServer* has a relation with the *CControllerBase* class which decides which control module will be processing the incoming commands.

Finally the transfer unit (packet) between the scripts and the control server is explained. Note that because we are working with scripts which usually have very good string handling support and no datatypes I decided that the values within the packet are always strings. Therefore in order to put a number n into the packets payload the number would first have to be converted to a string. The maximum size of the packet is 1MB. The structure of the packet is the following:

Field	Size
Packet header	10Byte string
Command	10Byte string
Data length	10Byte string
Data	Data length byte long string

To provide support for at least one scripting language I provided PHP support class which takes care of the server-script protocol. The class provides two basic functions:

```
public function WriteCommand($cmd, $len, $data);
```
Writes a command to the shared memory block.

```
public function ReadCommand()
```
Waits for a server command header and when it receives it it returns the received packet.

Using this class the users can now implement a control module specific class which can be used with some specific module if its loaded. The first operation what such class should do (in its constructor) is make a handshake with the loaded module to make sure that the proper module is loaded. I provided such classes in PHP language for both of the controlling modules which I implemented: the *BS001I2C* and *BS001CAN* classes for controlling I2C and CAN buses.

Once the module specific class is provided users can start making libraries on top of this class to provide some advanced control/emulation operations. I provided one such library with the Bus-Spy environment. It is the 24c128 i2c eeprom library which encapsulates the basic eeprom operation into a PHP class and allows its user to do whatever he or she wants with the real eeprom physically connected to the controled i2c bus. The class which provides all the 24c128 functionality is called *BS001_24C128*.

In the end of this section I provide a small snippet of code demonstrating the use of the 24c128 php library. The code reads all the data from the eeprom and nicely prints them on the screen. The sample output from this script can be seen on the figure 7.8. As you can see valid data were loaded from the eeprom.

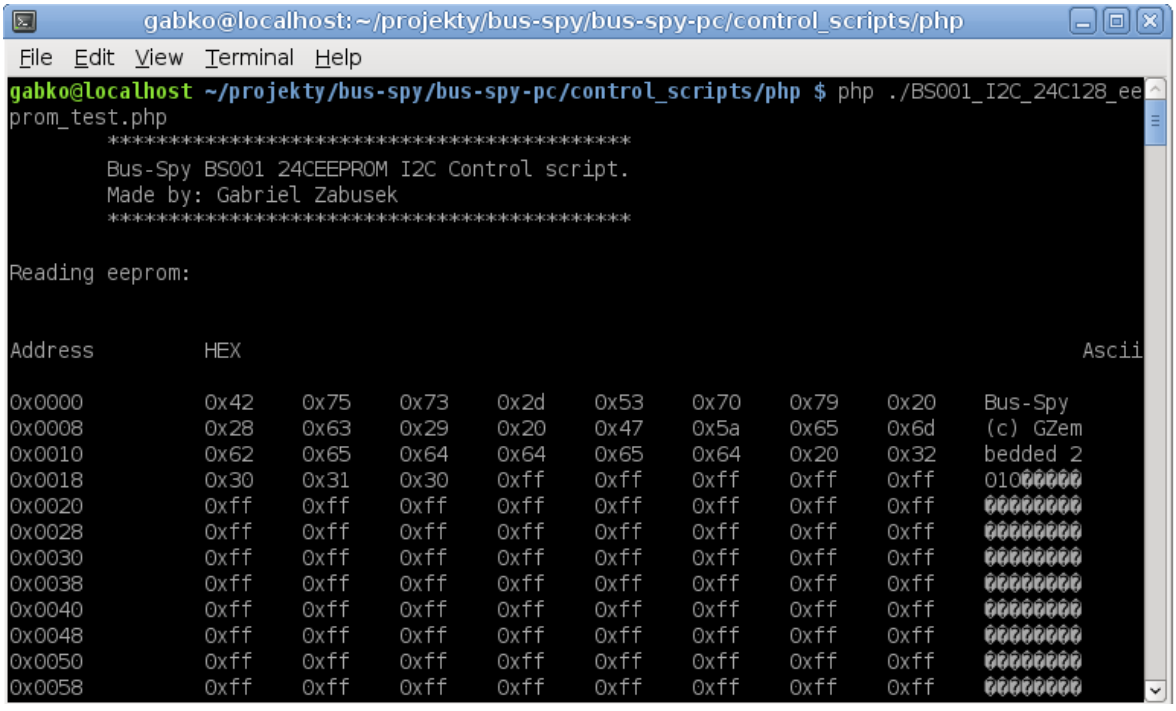


Figure 7.8: Sample output from the i2c eeprom controlling script.

```php
<?php

include "BS001_I2C_24C128_EEPROM.php";
define('BYTES_LINE', 8);

// create new eeprom instance with the shared memory key 0x51012497
$eeprom = new BS001_24C128(0x51012497);

printf("Reading eeprom:\n");

// header
printf("\n\nAddress\t\tHEX");
for($i=0;$i<BYTES_LINE;$i++)
    printf("\t");
printf("\tAscii\n\n");

// go through the eeprom addresses
for($i=0x0000; $i<0x3FFF; $i+=BYTES_LINE)
{
    $asciival = "";
    printf("0x%04x\t", $i);

    // read the data from the eeprom
    $rd = $eeprom->read_data($i, BYTES_LINE);

    // pretty print the data in hex
    for($j=0; $j<BYTES_LINE; $j++)
    {
        printf("\t0x%02x", $rd[$j]);
        $asciival .= chr($rd[$j]);
    }

    // print the data in ascii
    echo "\t$asciival\n";
}

?>
```

7.8 I2C and CAN bus control and analysis modules

In this section I will describe the operation of the control and analysis modules which I created. They are the *CAnalyserI2C*, *CAnalyserCAN*, *CControllerI2C* and *CControllerCAN* modules and are capable of analysing and controlling the i2c and CAN buses in their basic form. I will now describe the principal of operation of every one of them.

CAnalyserI2C is a zero knowledge analysis module to analyse the i2c bus. The settings window of this module is on the figure 7.9. As you can see the initialization

parameters for this module are these:

- Trigger
- Sampling frequency
- Data pin
- Clock pin
- Trigger pin

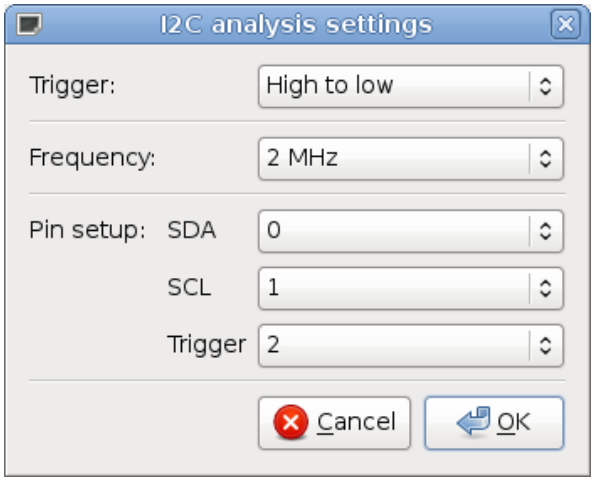


Figure 7.9: The settings window of the *CAnalyserI2C* module.

These parameters are encapsulated in the custom fields of the INIT Bus-Spy packet as displayed on the figure 7.10.

When the start of the i2c analysis operation is requested the working window (displayed on the figure 7.11) of this module is displayed and the module is started[3]. The module then starts its data collector and waits for the data from the device. When a packet is received the module requests it from the data collector and starts analysing the raw data. The data payload is split into 4 Byte chunks (unsigned integers). The even chunks contain data line samples and the odd chunks contain the clock line samples. The meaning of the data within the chunks is the following: The most significant bit represents sample at time n and the least significant bit represents sample at time $n + 31$. If chunk k contains the data line samples beginning at time i then the chunk $k + 1$ contains the clock line samples which also begin at time i.

	1B	1B	1B	1B
Custom field 1	Trigger code	Freq. code	Clock pin	Data pin
Custom field 2	Trigger pin	X	X	X

Figure 7.10: The custom fields used during i2c analysis initialization.

[3]Using the start() method inherited from QThread

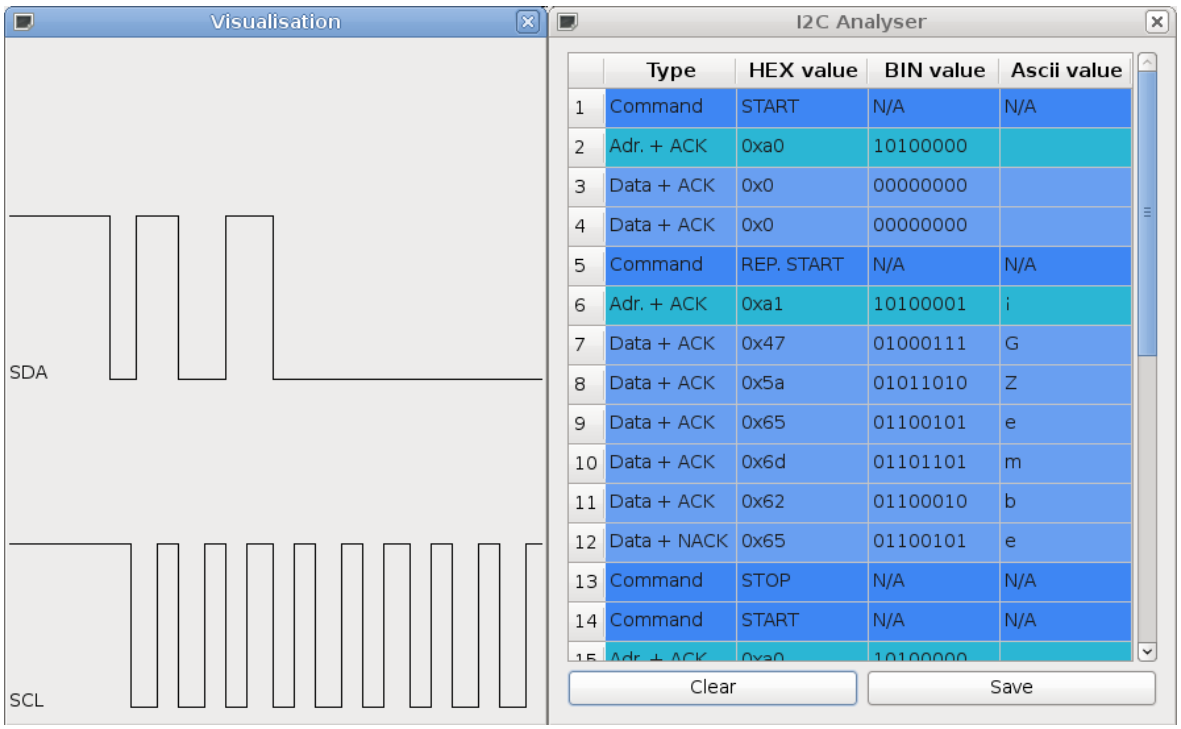


Figure 7.11: The working window of the *CAnalyserI2C* module.

CAnalyserCAN is a with knowledge CAN bus analysis module. The settings window of this module is on the figure 7.12. The only initial parameter is the bus' bitrate.

The beginning of the operation is equal to the beginning of the *CAnalyserI2C* module. That means that the working window is opened and the started module waits for some data after it starts its *CDataCollector* instance.

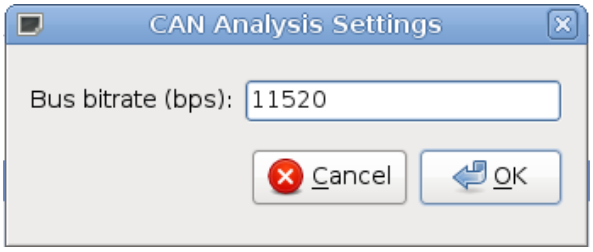


Figure 7.12: The settings window of the *CAnalyserCAN* module.

Once a packet arrives the module starts extracting the CAN packets from the data payload. The custom field 1 contains the number of CAN packets inside. Each CAN packet takes 32 Bytes within the data payload area of a Bus-Spy packet[4]. There can be two types of CAN packets:

- Data packet
- Error packet

The structures of these packets are on the figures 7.13b and 7.13a. When the CAN packet is extracted it is displayed on the working window according to the type of the packet. The working window is displayed on the figure 7.14.

[4]Which means there can be up to 16 CAN packets stored within one Bus-Spy packet.

1B	1B	1B	1B
X	CODE	DIR	Err. Type
Reserved			
Reserved			
Reserved			
TIME			
Reserved			TYPE
Reserved			
Reserved			

(a) Error CAN packet

1B	1B	1B	1B
X	RTR	IDE	DLC
ID			
DATA[0..3]			
DATA[4..7]			
TIME			
Reserved			TYPE
Reserved			
Reserved			

(b) Data CAN packet

Figure 7.13: Error and Data CAN packets within the Bus-Spy packet data payload.

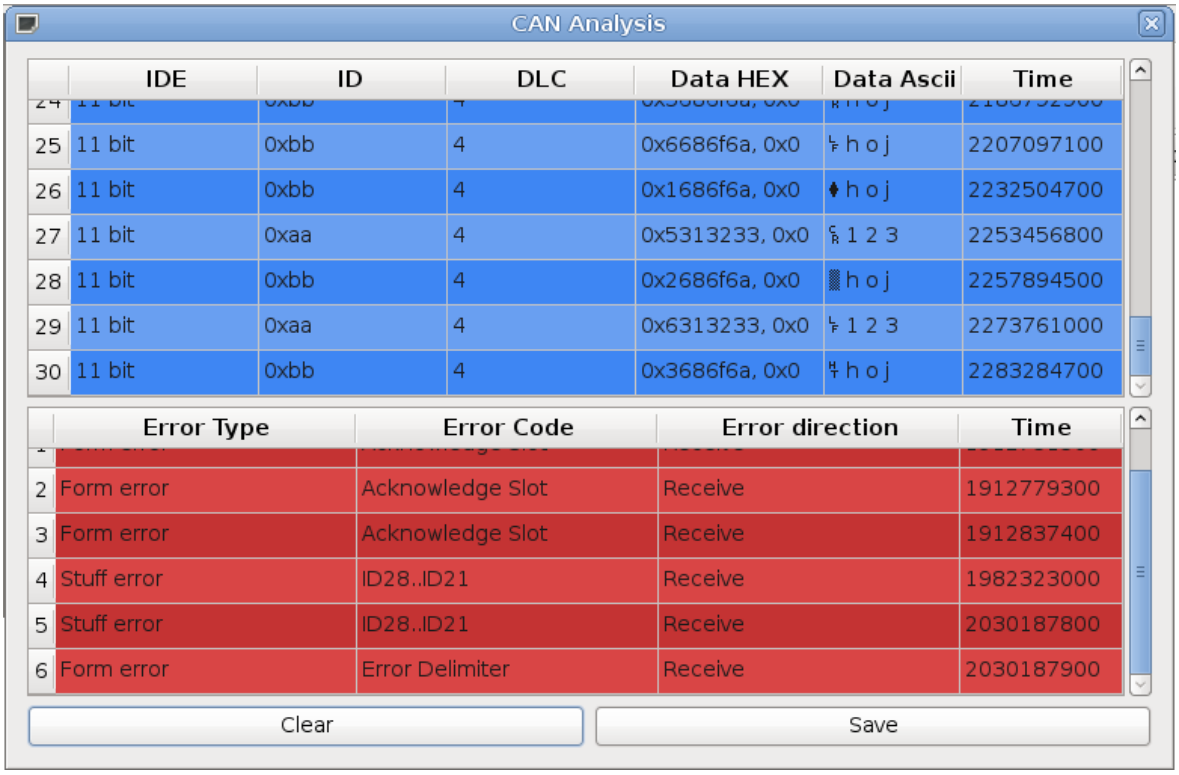


Figure 7.14: The working window of the *CAnalyserCAN* module.

CControllerCAN and CControllerI2C are script driven modules for controlling the i2c and CAN buses. They work in a very similar way and therefore I will describe them together.

When the operation starts the working window of the module is opened and the modules instance is assigned to the control server. After that the module is waiting for the control server to call its ProcessCommand() method. As previously mentioned there is always at most one control module assigned to the server.

Whenever some command from an external script arrives the server passes is to the current control module which in turn processes it and returns some reply command/packet. This reply command is then passed to the external script by the control server.

The initialization parameters for the *CControllerCAN* module are exactly same as for the *CAnalyserCAN* module and can be seen on the figure 7.12. The settings window of the *CControllerI2C* module is displayed on the figure 7.15. As you can see the initialization parameters are the following:

- Device type (Master/Slave)
- Master clock speed (used in case the device type is Master)
- Slave address (used in case the device type is Slave)

These parameters are passed to the device in the custom fields of the INIT packet. The custom field 1 contains the device type and the custom field 2 contains either the clock speed or the slave address according to the custom field 1.

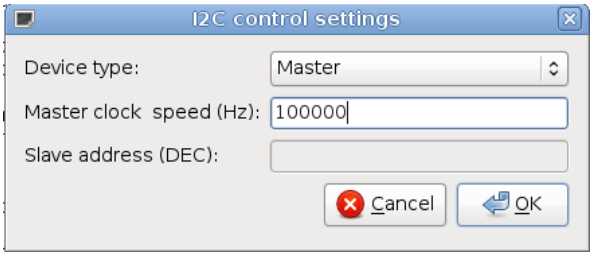


Figure 7.15: The settings window of the *CControllerI2C* module.

The working windows of these modules are only used as a loggers of the communication between the script and the device. There are PHP classes provided which offer the basic operations to work with these modules. They are the *BS001CAN* and the *BS001I2C* classes and provide operations for all supported commands by their modules as described in the sections 6.5 and 6.7. The working windows including the control scripts and their outputs are displayed on the figures 7.16 and 7.17.

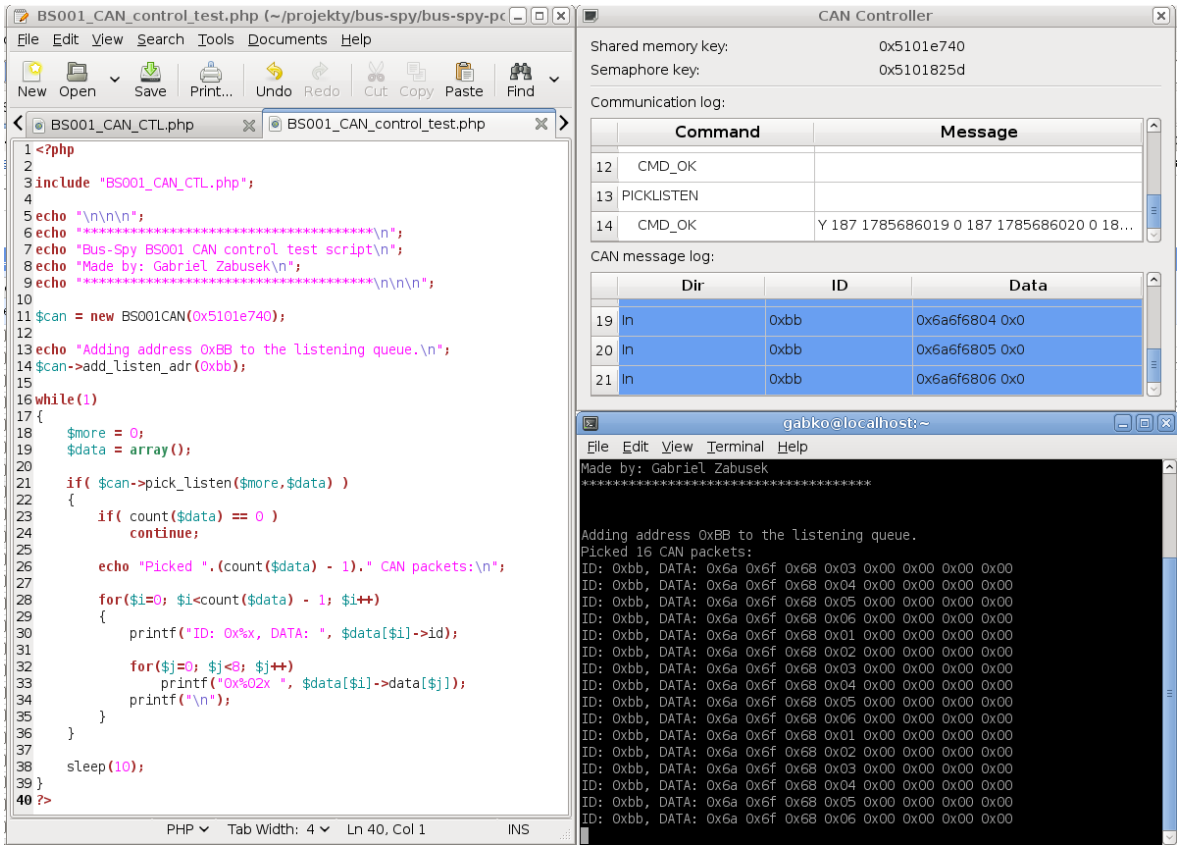


Figure 7.16: The working window of the *CControllerCAN* module.

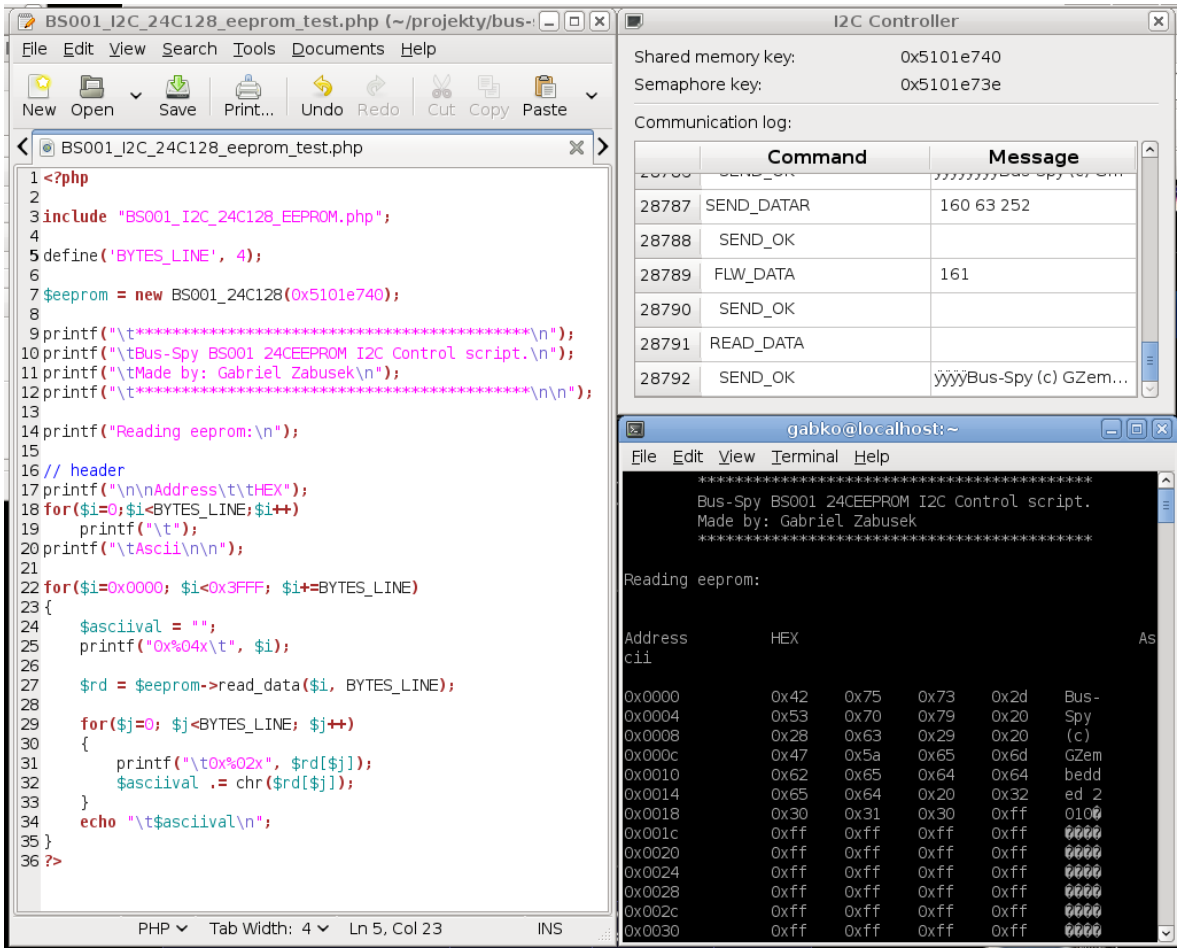


Figure 7.17: The working window of the *CControllerI2C* module.

7.9 Principle of operation

In this section I will briefly describe the principle of operation of the pc application from when it is started.

When the Bus-Spy application starts it prepares the main window to the initial state. Next, it creates the instances of all supported device modules (which in turn create the instances of all the control and analysis modules that they support). Finally, it creates the instance of the control server and waits for user actions.

The main application window is designed in a way that the very first action which must be done is selecting some device module. When the device selection was made the transfer bus and work type selections are filled according to what the selected device supports. As soon as the work type is selected by an user the bus selection gets filled with all bus names on which the previously selected device supports the selected work type.

Whenever some module is selected and it supports the settings window the settings button for the appropriate module becomes available. After clicking this button the modules settings window shows up.

When all the parameters are set the application is ready to connect with the device. Once the connection is requested the application checks the transfer bus and control/-analysis module settings by calling the accept() methods of their settings dialogues. If the settings are ok the connection is open and the INIT packet is sent to the device. If

the device replies with the INIT_OK command the start button becomes available.

Finally, when the start of the operation is requested the application opens the working window of the loaded control/analysis module and sends the START packet to the device. At that point the control is passed to this window until the disconnect action is requested.

Chapter 8

Summary and conclusion

In the previous chapters the existing solutions and possible approaches were analysed and detailed description of the created environment was provided. In this chapter I will summarize this work.

The original requirements of this work were to create a modular environment for controlling and monitoring digital buses. The software portion of the environment was supposed to support multiple commonly used operating systems and control and monitoring of i2c and CAN buses in their basic form.

All the requirements were successfully accomplished. The PC application runs on Windows, Linux and can be easily ported to MacOS X with only few changes to the source code. It is completely modular and adding new modules is quite comfortable. There are five type of modules which are currently supported:

- Bus control modules
  - Script driven
  - Manually controlled
- Bus monitoring/analysis modules
  - With knowledge analysers
  - Zero knowledge analysers
- Transfer bus modules
- Device modules
- Special modules

All the modules have built-in support for multi threading. The design of the transfer bus modules[1] allows easy and transparent separation of implementations for different operating systems. The design of the control modules allows layered structure of the modules which means that one control module can work on top of another one and its work is controlled by the events coming from this module.[2] Both designs are based on well known and popular design patterns.

The following final software modules were implemented in this work:

[1] Based on the Bridge design pattern.

[2] The event mechanism is based on the Observer design pattern.

- I2C (Zero knowledge) Analyser module
  - Non intrusively monitors i2c bus and displays all the data to the user.
  - Provides visualisation of the electrical values on the data and clock lines.
- I2C (Script driven) Controller module
  - Allows complete control of i2c bus.
  - Supports master and slave modes.
  - PHP classes for basic operations provided.
  - Fully functional 24C128 emulator / controller class in PHP script provided.
  - Two example scripts provided: One controls real eeprom physically connected to an i2c bus and one creates a virtual eeprom and acts as a real one on a real i2c bus.
- CAN (With knowledge) Analyser module
  - Non intrusively monitors CAN bus and displays all the data to the user.
  - Provides detailed information in case of bus error.
- CAN (Script driven) Controller module
  - Allows complete control of CAN bus.
  - PHP classes for basic operations provided.
  - One example script provided which listens to predefined IDs and displays them to an user.
- Ethernet transfer bus module
  - Classes with Windows and Linux implementations provided.
- BS001 device module
  - Encapsulates all the functions supported by the BS-001 device created in this work.

Lastly a fully functional prototype device[3] which supports all the aforementioned modules was created.

[3]Named BS-001.

Appendix A

BS-001 Bus-Spy device user documentation

This chapter serves as an user documentation for the example device BS 001 created during this work. Note that the device is currently a first prototype and therefore the 'user interface' is not very user friendly as can be seen on the figure A.1.

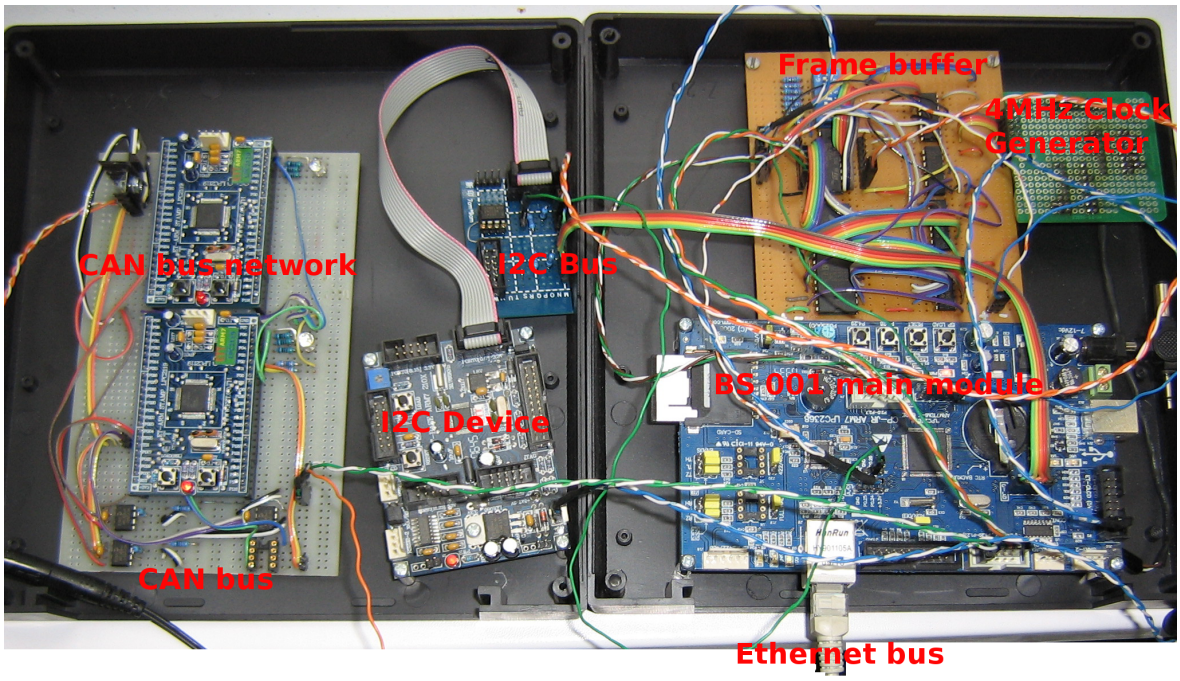


Figure A.1: The hardware portion of the Bus-Spy environment including two test devices.

There are several labelled groups of wires coming out of the device. The labels and their purpose is the following:

Label	Description
Input pin 0	Used for (i2c) zero knowledge analysis.
Input pin 1	Used for (i2c) zero knowledge analysis.
Trigger pin	Used as a trigger for (i2c) zero knowledge analysis.
CAN bus	Used for with knowledge CAN bus analysis and script driven CAN bus control.
I2C bus + 3V3 voltage	Used for script driven I2C control. Optionally +3.3V connection can be used as the bus voltage.
5V0 voltage	Can be used to power up a 5V device so that the ground levels on the BS001 and analysed device will be the same.

When the device is powered on it needs to be hard reset first so that the PHY layer of the Ethernet block is properly initialized. After that the device is ready to operate.

In order to communicate with a PC the device needs to be connected either to a hub or switch on the PCs network or directly with the PC using a crossed Ethernet cable. The default address for the device is 192.168.1.55 with default gateway set to 192.168.1.1 and having mask 255.255.255.0. These settings can be changed in the service mode which is controlled from the PC via standard RS232 (serial) cable. The service mode is entered if the service mode button was pressed during boot.

In the following sections I describe what has to be done in order to start some operation without problem. The device supports the following operations:

- CAN bus with knowledge analysis
- I2C zero knowledge analysis
- CAN bus script driven control
- I2C bus script driven control

A.0.1 CAN analysis

In order to analyse a CAN bus simply plug in the pair of wires labelled as *CAN bus* into the appropriate pins of a CAN transceiver connected to the CAN bus which you want to analyse. Each wire from from the CAN bus couple is labelled with the appropriate signal name[1].

A.0.2 I2C analysis

To analyse an I2C bus you will have to use the wires labelled as *Input pin 0*, *Input pin 1* and *Trigger pin*. Connect the input pins to SDA and SCL pins of the i2c bus (The concrete connection than has to be replicated in the Bus-Spy utility using the settings window of the I2C analysis module.) and the trigger pin to a connection which you want to scan for the trigger event. The analysis can start at this point.

A.0.3 CAN control

Getting ready for CAN control is identical to the CAN analysis. Simply plug in the bus wire labelled as *CAN bus* to the CAN transceiver connected to the bus which you want to control. Once done the CAN bus control operation can start.

[1] CAN-H and CAN-L or rx,tx respectively

A.0.4 I2C control

In order to control an I2C bus you need to use the wire labelled as *I2C Bus + 3V3*. The bus wire contains four connections:

- +3.3V (Brown)
- SDA (Red)
- SCL (Orange)
- GND (Yellow)

Connect the SDA and SCL wires to the i2c bus which contains pull up resistors. The power connection can be used to power up the circuits connected to this i2c bus. At this point everything is ready to control the bus via Bus-Spy application.

Appendix B

Bus-Spy user documentation.

This chapter contains the user documentation of the Bus-Spy PC application. The requirements to run Bus-Spy with all the features enabled are the following:

- Window or Linux powered computer
- Bus-Spy compatible device connected to the computer.
- 2 core 1.5GHz and more CPU
- 512MB RAM
- Ethernet
- Any version of PHP compiled with shmop (shared memory) support.

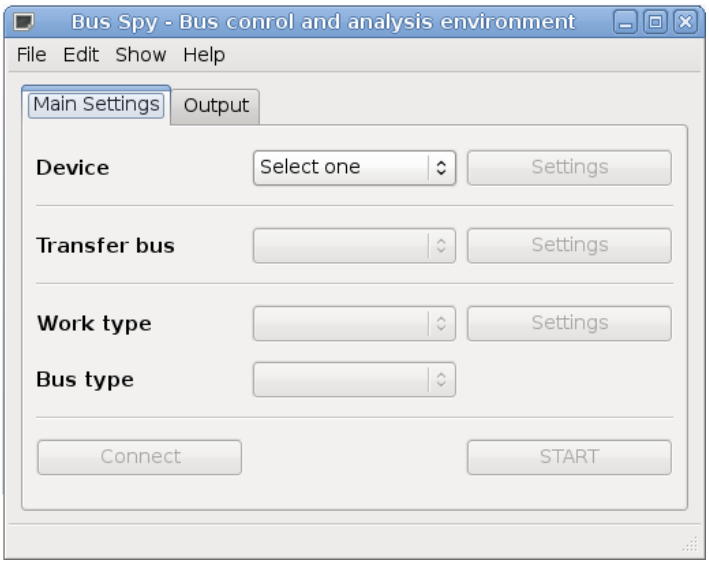


Figure B.1: The main Bus-Spy window after start.

The operation of Bus-Spy is very simple. Once the application is started by running the bus-spy-pc executable[1] the main window is displayed. This window (right after start) is also displayed on the figure B.1. After the application was started the following steps (in order) are necessary:

1. Select some device from the device combo box.
2. If the device has settings window (the settings button near the combo box became available) open it by pressing the settings button and set the desired settings.
3. Select some transfer bus from the transfer bus combo box.
4. If the transfer bus has a settings window open it by pressing the settings button and set the transfer bus settings[2]

[1] On a Windows computer its bus-spy-pc.exe

[2] In case of Ethernet it is the interface to use and the IP address of the device. This is displayed on the figure B.2

5. From the work type combo box select the type of operation which you desire; either control or analysis.

6. From the bus type combo box select the type of the bus which you want to control or analyse (monitor).

7. If the selected module has a settings window (the settings button near work type combo box becomes available) set all these settings.

8. At this point the Connect button becomes available, when you click on it the application will try to open a connection with the device.

9. If the connection was successfully opened the START button becomes available, when you press it the operation starts immediately.

10. If the selected control/analysis module supports working window it opens automatically and takes over the control.

11. All the working windows provided with this work are straight forward and are only used as a 'displayers' of data. In case of other future modules refer to the user documentation of the future module.

12. Once you are done with the operation you can press the Disconnect button and start over from 1).

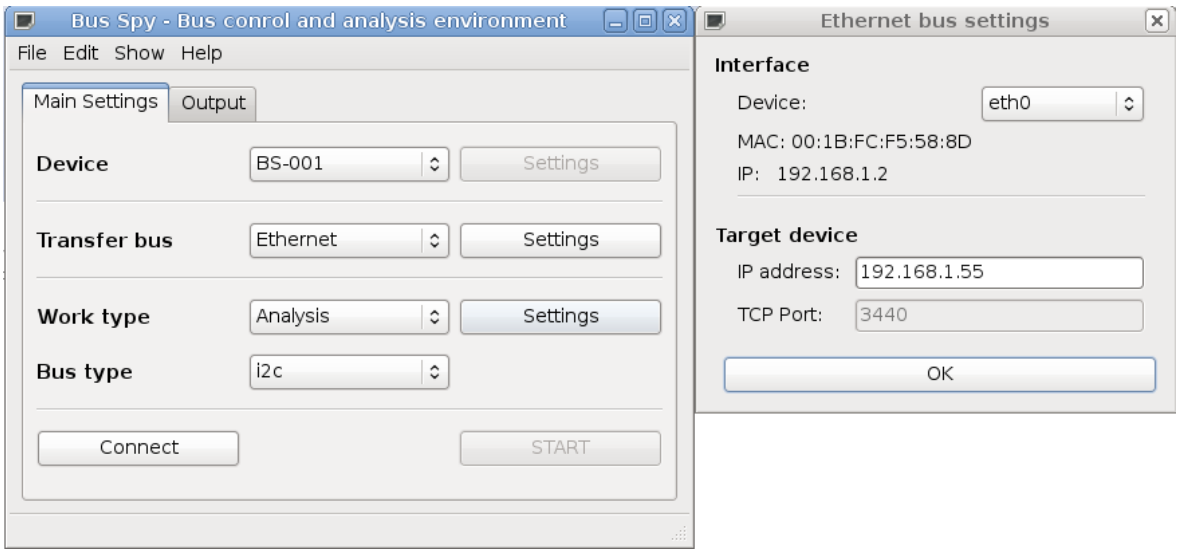


Figure B.2: The main window with some selections made. Ethernet transfer bus settings window.

Appendix C

Bus-Spy programming fundamentals.

In this chapter I will describe the fundamentals which should be known by someone interested in upgrading and developing the Bus-Spy environment. Note that detailed source code documentation[1] can be generated using doxygen in the root directory.

C.0.5 Code structure

The code of the Bus-Spy environment is written in C, C++ and assembler programming languages. To compile the code I used the GNU compiler utilities[2]. The support scripts for controlling a bus are written in PHP language. The structure of the source code folders is the following:

- **bus-spy** (the root folder of the source code)
  - **bus-spy-pc** (PC application related source code)
    - **src** (Source of the actual PC application)
      - Root of this directory contains the code for the window subclasses.
      - **common** (Contains basic definitions and things common to the PC and embedded source code)
      - **device** (Device modules related source code)
      - **module** (Base classes for modules and basic module related code)
      - **target_bus** (Target bus related source code such as control and analysis modules)
      - **transfer_bus** (The base classes for transfer bus modules and the actual transfer bus modules)
    - **ui** (standard QT descriptors of the windows)
    - **control_scripts** (Bus control related source code)
      - **php** (Control scripts in PHP language)
  - **bus-spy-device** (Hardware and test devices related source code)
    - **lpc2368_ca_device** (BS-001 device source code)
    - **test_devices** (Source code of the test devices)

[1] In html, LaTeX, and xml formats
[2] gcc, ld, etc.

- **lpc2103_eeprom** (Source code of the test device used for analysis of the I2C bus)
- **lpc2119_can_node_A** (Source code of the test device A used in the CAN 'led network test device' which is used for CAN control and analysis)
- **lpc2119_can_node_B** (Source code of the test device B used in the CAN 'led network test device' which is used for CAN control and analysis)

To compile the source code of the BS-001 device or any other test device simply enter the directory and type *make*. Once the source is compiled you can use *make program* to flash the firmware into the device. This assumes you have the *lpc21isp*[22] or the *armflash*[23] flashing utility.

The source code of the Bus-Spy is compiled in exactly the same way (using *make*) except that the Makefile has to be generated first by QT framework support tools. To do this simply execute *qmake* in the bus-spy-pc directory and once that command finishes the Makefile will be generated.

C.0.6 Adding new modules

In this section I will briefly describe how to add a new module to the environment.

Device modules

To add a new device module simply create a class which inherits from the *CDeviceBase* class and implement the following functions:

```
CDeviceBase(string name, string description);
```

The constructor should call the base classes constructor and create the modules instances at least.

```
virtual bool TransferBusSupported(transfer_bus_t type) = 0;
```

Transfer bus support.

```
virtual bool InitTransferBus(transfer_bus_t type) = 0;
```

Initializes the requested transfer bus.

```
virtual bool Connect() = 0;
```

Connects to the previously initialized transfer bus.

```
virtual bool WorkOnBusSupported(work_type_t work, bus_type_t bus) = 0;
```

Determines work on a bus support.

```
virtual bool WorkSupported(work_type_t work_type) = 0;
```

Determines work type support.

```
virtual CAnalyserBase * GetAnalyser(bus_type_t bus_type) = 0;
```

Returns the requested analyser for bus type in parameter.

```
virtual CControllerBase * GetController(bus_type_t bus_type) = 0;
```
Returns the requested control module for the bus type in parameter.

```
virtual CTransferBus * GetTransferBus() = 0;
```
Returns the initialized transfer bus module.

```
unsigned int GetNumInputPins();
```
Returns the number of input pins of the device which can be used for zero knowledge analysis.

```
bool HasSettings();
```
Determines whether the module has settings window.

```
void ShowSettings();
```
Shows the settings window if it has any.

```
bool HasWindow();
```
Always returns false for device modules.

```
void ShowWindow();
```
Always returns false for device modules.

```
void CloseWindow();
```
Does nothing for device modules.

```
bool GetSettingsAsCustomFields(
                        custom_field_t (*fields)[NUM_CUSTOM_FIELDS],
                        work_type_t work_type,
                        bus_type_t bus_type,
                        transfer_bus_t transfer_bus_type);
```
Returns the modules settings as custom fields of the Bus-Spy packet (see 5.4).

Once your device module class is ready there is one more thing which needs to be done. The instance of this class needs to be created anywhere in the constructor of the *MainWindow* class located in the *mainwindow.cpp* file.

C.0.7 Transfer bus modules

To add a new transfer bus module you need to create a class which inherits from the *CTransferBus* base class. Next you need to implement at least one[3] implementation class for this module by creating a class which inherits from the *CTransferBusImpl* base class. Both of these classes have the same methods but the *CTransferBus* class also inherits methods from the *CModuleBase* class since it is also a regular module. The sibling of the *CTransferBus* should implements all its vitual methods it inherited. The implementation of all the functions inherited from the *CTransferBus* class is simply a call to the method with the same name which is member of the current implementation class. This implementation class is passed to this one in the constructor and stored as a private member. Then it needs to implement the window related functions inherited from the *CModule*Base* classes which are identical to the ones described in the C.0.6.

[3]But typically two will be provided. One for Windows and one for Linux operating system.

C.0.8 Control modules

To create a new control module simply create a class which inherits from the *CControllerBase* base class and implement at least these methods:

`Constructor`
The constructor typically sets the windows availability using the methods inherited from the *CModule*Base* classes and calls the *CAnalyserBase* constructor which takes care of registering the module.

```
control_command_t ProcessCommand(control_command_t &cmd);
```
Processes the commands from the control server.

```
bool GetSettingsAsCustomFields(
                        custom_field_t (*fields)[NUM_CUSTOM_FIELDS],
                        work_type_t work_type,
                        bus_type_t bus_type,
                        transfer_bus_t transfer_bus_type);
```
Returns the settings of this module as a custom fields of Bus-Spy packet.

Finally you need to implement the window related functions as described in the C.0.6.

C.0.9 Analysis modules

To create a new analysis module you need to create a class which inherits from the *CAnalyserBase* class and implement the following methods at least:

`Constructor`
The constructor typically sets the windows availability using the methods inherited from the *CModule*Base* classes and calls the *CAnalyserBase* constructor which takes care of registering the module.

```
void run();
```
This is the thread procedure which is executed after start() function inherited from the *QThread* class. It is called when the operation of this module should start.

```
bool GetSettingsAsCustomFields(
                        custom_field_t (*fields)[NUM_CUSTOM_FIELDS],
                        work_type_t work_type,
                        bus_type_t bus_type,
                        transfer_bus_t transfer_bus_type);
```
Returns the settings of this module as a custom fields of Bus-Spy packet.

Finally you need to implement the window related functions as described in the C.0.6.

Bibliography

[1] Andrew N.Sloss, Dominic Symens, Chris Wright: *ARM System Developer's Guide*, Elsevier, 2004, ISBN-13: 978-1558608740.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* , Addison-Wesley Professional, 1994, ISBN-13: 978-0201633610.

[3] Jasmin Blanchette, Mark Summerfield: *C++ GUI Programming with Qt 4*, Prentice Hall PTR, 2006, ISBN-13: 978-0131872493.

[4] Bosch, *CAN bus specification version 2.0*, 1991

[5] Philips semiconductors, *The I2C Bus specification version 2.1*, 2000

[6] Philips, *LPC2364/66/68 Data sheet Rev. 01*, Philips, 2006.

[7] Philips, *LPC23XX User manual Rev. 01*, Philips, 2008.

[8] ETT Team, *User's manual of CP JR ARM7 LPC2368.*

[9] Philips, *LPC2101/02/03 Data sheet Rev. 01*, Philips, 2006.

[10] Philips, *LPC2101/02/03 User manual Rev. 04*, Philips, 2009.

[11] ETT Team, *User's manual of ETT ARM7 LPC2103.*

[12] Philips, *LPC2119/2129/2194/2292/2294 User manual*, Philips, 2004.

[13] ETT Team: *User's manual of ETT ARM7 LPC2119 Stamp.*

[14] Philips, *AN10403 Connecting Ethernet Interface with LPC2000 V1*, 2007.

[15] Philips, *AN10576 Migrating to the LPC2300/2400 family V1*, 2007.

[16] Philips, *AN10406 Accessing SD/MMC Card using SPI on LPC2000 V3*, 2007.

[17] Philips, *AN10799 Porting uIP1.0 to LPC23xx/24xx V2*, 2009.

[18] Philips, *AN10674 NXP LPC2000 CAN driver with FullCAN mode*, 2009.

[19] Philips, *Creating your own CAN Network with the NXP LPC2300 series (by ARM)*, 2009.

[20] Lukas Koch, *http://ast.m-faq.de/.*

[21] Thomas Scherrer, *http://www.z80.info/uexosc.htm Crystal oscillator circuits.*

[22] lpc21isp, *http://sourceforge.net/projects/lpc21isp/*.

[23] armflash, *http://code.google.com/p/armflash/*.

[24] LeCroy VBA104Xi, *http://www.lecroygmbh.com/*

[25] I2C/SPI Protocol analyser and Host Adapter, *http://www.totalphase.com/products/beagle_ism/*

[26] CAS-1000-I2C/E, *http://www.corelis.com/products-bus-analyzers/Bus_Analyzer_I2C_CAS-1000-I2C-E.htm*

[27] MDFly, *http://www.mdfly.com/*

[28] FriendlyARM mini2440, *http://www.friendlyarm.net/products/mini2440*

[29] ETT LPC2368 controller, *http://www.futurlec.com/ARM2368_Controller.shtml*

[30] GNU compiler tools, *http://gcc.gnu.org/*

[31] Gedit text editor, *http://projects.gnome.org/gedit/*

[32] QT GUI Library, *http://qt.nokia.com/products/*

[33] Doxygen, *http://www.doxygen.org/*

[34] Latex, *www.latex-project.org/*

[35] uIP TCP/IP Stack, *www.uip.com/*

[36] PHP Hypertext preprocessor, *www.php.net/*

[37] CAN bus description, *http://www.interfacebus.com/Design_Connector_CAN.html*

List of Figures