Czech Technical University in Prague

Faculty of Electrical Engineering

Bachelor thesis

TestLab - ProfiNet Network Tester

Tomáš Davidovič

Project supervisor: Ing. Martin Novotný

Study program: Electronics and Computer Science & Engineering

Branch of study: Computer Science and Engineering

June 2006

Acknowledgments

Foremost I would like to thank my colleague Vít Bernatík for his excellent work and cooperation in co-designing this project. I would also like to thank Ing. Martin Novotný for giving me the opportunity to participate on this project and to Ing. Jan Hušák and Ing. Jan Hvozdovič for their very good leadership.

Declaration – Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 13. července 2006 ……..

Tomáš Davidovič

Abstract

The tempLab system is designed to test PROFINET industrial network that's based on the 10Base-T Ethernet standard. The main functionalities are the option to independently break the RX and TX wires in an Ethernet cable and the option to use two LAN switches for an immediate network configuration change. The wire breaking allows simulation of communication interruption while the configuration change allows for example a short time connection of a measurement device or a generator. Both the breakers and the switches can be set completely independently in several start modes and with times defined in 21ns steps. Other offered functions are temperature measurement with up to sixteen thermal probes and turning up to sixteen tested devices on and off.

The whole tempLab device is connected by an USB cable to a controlling PC that controls all the functions. There are two applications dedicated to the function control. The first one is a GUI application that allows easy control of the tempLab's correct functionality and its easy calibration. The other one is represented by a support library for Python script language that will be used to write the PROFINET tests. Both applications communicated with the tempLab device via a common DLL library that can be in the future easily used for any other application that will need to communicate with the tempLab device.

Abstrakt

Systém tempLab slouží k testování industriální sítě PROFINET založené na 10Base-T Ethernetu. Hlavními funkcemi jsou možnost nezávisle přerušovat vodiče RX a TX v jednom Ethernet kabelu a možnost využít dvou síťových přepínačů k okamžité změně konfigurace sítě. Přerušování vodičů umožňuje simulovat výpadky komunikace na síti, zatímco změna konfigurace umožňuje například krátkodobé připojení měřících zařízení či generátorů. Oba přerušovače i oba přepínače větví lze nastavovat zcela individuálně v několika spouštěcích módech a s časy definovanými v krocích po 21ns. Dalšími nabízenými funkcemi jsou měření teplot až šestnácti teplotními sondami a vypínání a zapínání až šestnácti testovaných zařízení.

Celé zařízení je pomocí USB kabelu připojeno k řídícímu PC, které ovládá veškeré jeho funkce. K ovládání zařízení slouží dvě aplikace. První z nich je GUI aplikace umožňující snadnou kontrolu správné funkce zařízení a jeho snadnou kalibraci. Druhá je představována podpůrnou knihovnou pro skriptovací jazyk Python, ve kterém se budou psát samotné testy sítě PROFINET. Obě tyto aplikace komunikují se zařízením pomocí společné DLL knihovny, kterou lze v budoucnu snadno využít i pro další aplikace, které budou potřebovat komunikaci s tempLabem.

Content

1. Introduction

PROFINET [1] is an open Industrial Ethernet standard, electrically compatible with 10Base-T Ethernet. It is used in factory automation, to control various devices used in automated factories.

Such an environment generates a lot of electrical noise that can interfere with the signal on PROFINET cables and interrupt the communication. However devices connected by PROFINET need to operate continually, so there is a set of requirements concerning how the devices should react to communication interruptions of different lengths and timings.

This project deals with development of a solution that will allow to test whether various PROFINET devices conform to those requirements. The required functionality of the solution is shown on ***Figure 1***.

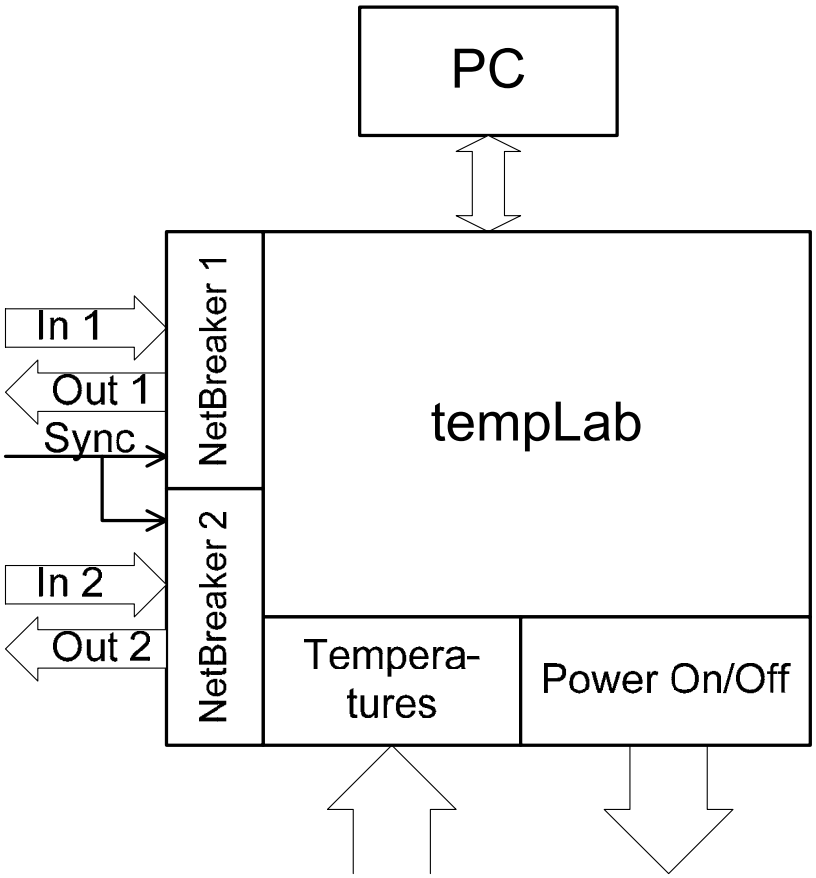

Figure 1 : Solution schematics

The tempLab device will be able to turn on and off up to sixteen tested devices and measure their temperature.

However, the main functionality is in the NetBreaker blocks. The In and Out ports are RJ-45 sockets for PROFINET cables. In normal state, each In is connected to its corresponding Out, so the signal goes through uninterrupted. For tests it must be

possible to independently interrupt the RX and TX wires in each of the cables for predefined times.

For timing of the NetBreaking (interruption) please see ***Figure 2***. For thorough testing it is necessary to control not only the length of NetBreaking (time T3), but also to control its start. For this we will use PROFINET's synchronization signal (Sync), that is generated periodically each 4ms, but can be as low as 0.5ms depending on configuration of PROFINET, and lasts for 1µs. Start of the NetBreaking is defined relatively to the rising edge of Sync (time T2). The last specified time value (T1) defines how long after the start command is received should the Sync signal be ignored, which is used to offset several independent NetBreakings.

The tests will be performed by scripts written in Python language running on a PC, so a Python object for communication with the tempLab device will have to be written. Another requirement was a user-friendly GUI application for testing functionality of the tempLab device. To avoid writing the whole communication twice, a communication library should be written, using Common Object Model (COM) as the COM technology allows DLL library functions to be called from virtually any language.

The initial development of the tempLab was to be made on an ATMega128 development kit (ATSTK500) by Atmel, using provided libraries for communication with PC over UART. The final version of the device should be using a dedicated Printed Circuit Board (PCB) and a contemporary communication standard, either USB or TCP/IP.

The assignment had been divided into two parts, I have been assigned with development of the control software for PC, my colleague [4] was assigned development of the tempLab device itself, both hardware and software.

Both of us had been assigned with research of possible solutions for the final platform. The main requirements were support of a contemporary communication standard and use of an AVR family MPU from Atmel, to ensure easy portability from the development kit.

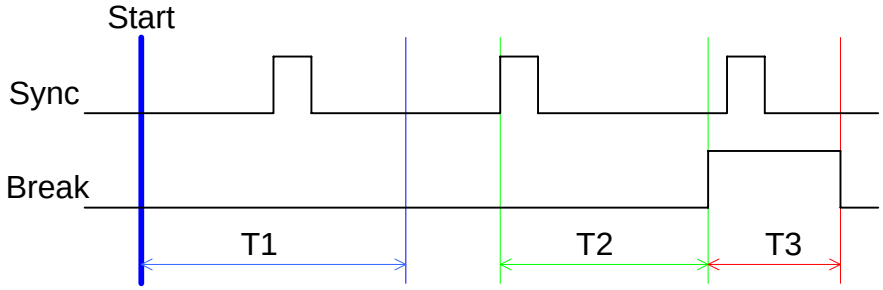

Figure 2 : NetBreaker times

2. Detailed technical specification

The specification is divided into requirements on the tempLab device itself and on the controlling PC software.

2.1 tempLab device

The device should conform to the following requirements:

- Be able to turn on and off up to 16 tested devices, using relays.
- Measure temperature in each of the 16 devices, one at a time, with tolerance ±1°C in range from -20°C to 90°C.
    - Pulse Width Modulation (PWM) thermometers SMT160-30 [18] suggested.
- Allow controlled NetBreaking of the communication on Ethernet cables:
    - Two pairs (numbered 1 and 2) of In and Out RJ-45 sockets.
    - Separate NetBreaking of RX and TX between the corresponding In and Out.
    - Starts NetBreaking in reaction to the rising edge of Sync signal, generated in intervals 0.5, 1, 2 or 4ms (depends on the configuration of PROFINET) and lasting for 1µs.
    - NetBreaking is defined by time values T1, T2 and T3, for their meaning see ***Figure 2***:
        - T1 defines how long after start will the Sync signal be ignored. Specified in milliseconds with no exact range of values required.
        - T2 defines how long after the Sync NetBreaking starts. Specified in microseconds, up to 4 milliseconds.
        - T3 defines how long will the net be disconnected (NetBroken). Specified in microeconds, ranges from 3 milliseconds to 60 seconds.
- Initial development using ATMega 128 Development Kit (ATSTK500) and UART communication.
- Final product implemented on a dedicated PCB, using contemporary communication standard based on AVR family MPU.

2.2 PC software

- Control of the device and basic functionality checks must be available via a GUI application using Microsoft Foundation Class (MFC) technology.
    - The application should minimize into system tray instead of the Windows taskbar.

- Python test scripts have to have an easy-to-use interface for communication with the tempLab device.
- Communication with the tempLab device is to be implemented as Common Object Model (COM) Dynamic Link Library (DLL), using Microsoft's Advanced Template Libraries (ATL).
- Offered functions must include:
  - Turning the tested devices on and off:
    - **SetRelay**(ID, On|Off) - sets state of a single device.
    - **GetRelayState**() – gets states of all devices
    - **SetRelayState**(x) – sets states of all devices, format of the input unspecified
  - Measuring the temperature:
    - **GetTemp**(ID) – gets temperature from the specified thermometer
    - Calibration functions if implementing calibration proves necessary.
  - NetBreaker:
    - **Break**(RX1|RX2|TX1|TX2, T1, T2, T3) – sets times for NetBreaking RX respectively TX wire in the respective NetBreaker.
- Both GUI and DLL are to be written in Visual C++ SP6; the Python script should be able to run on version 2.3.4 with win32 libraries installed.

3. Research

The first assignment was research of possible solutions for the final platform. I have focused mainly on general availability and communication with PC, but I also had to take into consideration the device's functionality and the required precisions.

Control of relays for turning the tested devices on and off had no requirements concerning frequency of the MPU.

The implementation of NetBreakers hadn't been clear at that point, but it was assumed that with required times in microseconds, MPU frequency 6 MHz or higher should be enough.

The suggested PWM thermometers had a specified measurement error ±0.7°C. Therefore measurement of the produced PWM signal should introduce error no larger that ±0.3°C, to fit into the required precision of ±1°C. The temperature is computed from duty cycle (DC) of the PWM signal using this formula:

$$t = \frac{DC - 0.32}{0.0047}[°C] \quad (1)$$

The frequency of the PWM signal can range from 1 kHz to 4 kHz and my colleague's preliminary research showed it would take six clock cycles between two samples of the PWM signal.

The worst case scenario therefore is a 4 kHz PWM signal with the rising edge recognized immediately, the falling edge with six cycle delay and the next rising edge (marking the end of the whole DC measurement) again recognized immediately.

The maximal deviation of DC measurement therefore is:

$$\Delta DC = \frac{\frac{6}{f}}{\frac{1}{4000}} = \frac{6 \cdot 4000}{f} = \frac{24000}{f} \quad (2)$$

And maximal deviation of temperature measurement is:

$$\Delta t = \frac{\Delta DC}{0.0047} = \frac{\frac{24000}{0.0047}}{f} = \frac{5106382}{f}[°C] \quad (3)$$

This means that for the required measurement precision we need either frequency at least 17 MHz or a different measurement method.

3.1 TCP/IP Solutions

I have found three possible solutions using Ethernet for communication between PC and the device. The first two solutions are very similar and use software stack implemented in ATMega128 MPU. The third solution uses a dedicated TCP/IP chip and can be used with virtually any MPU.

3.1.1 Charon II and Ethernut

Charon II [2] and Ethernut [3] are two very similar software based solutions for Ethernet connection. Both come as either a final board or as a set of free-to-download PCB schematics. The schematics however include a RTL8019AS chip (Full-Duplex Ethernet Controller) by Realtek that proved to be hard to obtain in small quantities.

Both solutions run on ATMega 128, which would mean little or no trouble with porting the code from our development kit to the final device. As there is no direct hardware support for TCP/IP in the ATMega chip, the whole TCP/IP support is managed by an operating system NutOS. In addition, the OS allows reprogramming the device over Ethernet. However, is also consumes a lot of memory and presumably a lot of clock cycles as well.

The OS could introduce unpredictable interrupts and therefore unreliable time counting, which is not very good for an application based on precise time measurement.

Preventing this would require either a lot of time studying the NutOS code, or buying a development board just to test what happens if we turn off interrupts for up to one minute (the T3 time).

Pros:

- Can be re-programmed over the Ethernet.
- Is easy to attach to the company network.
- Designing our own PCB based on the schematics should be easy.

Cons:

- The device doesn't need an OS for anything but the Ethernet communication.
- The OS and its software TCP/IP stack consume a lot of resources for functionality we don't need (most of the OS functions).
- Obtaining the RTL8019AS chip in desired quantities could be hard as we hadn't found anyone selling less than a hundred pieces while we wanted just two.

3.1.2 W3100A based solution

W3100A [5] chip by the Korean company WIZnet is a hardwired TCP/IP chip, with support for all the protocols we would need (TCP, IP, MAC) and also some that might eventually become useful in the future (UDP, ICMP, DLC and ARP).

The chip implements a hardware stack, supports up to four independent connections, contains its own 16kB data buffer, can communicate with a MPU over either a bus or an I^2C serial interface.

While the free sample codes are for the 8051 MPU family, the company also offers an evaluation board with ATMega 128 [17] that comes with AVR sample codes and their technical support was very forthcoming about making the AVR codes available

without even buying the evaluation board. Also, the evaluation board is very well documented which should make designing our own dedicated PCB quite simple. The chip itself costs about 25USD.

Anyway, the evaluation board would probably prove to be necessary for the development and testing of communication.

Pros:

- Hardware stack means there will be almost no requirements for microprocessor resources.
- Available sample codes should be a good base for our own communication functions.
- The chip is not tied to any specific MPU, which allows high flexibility concerning the desired target frequency.
- A well documented board simplifies its integration into our own PCB design.

Cons:

- It might be necessary to port some of the 8051 code to AVR.
- The development board with ATMega 128 is quite expensive (over 300USD at the time of decision making).
- It is not a one-chip solution as it requires the MPU and the W3100A chip.

3.2 USB solutions

I found three suitable USB solutions. The first solution is a simple USB-to-UART converter that can be combined with any MPU; the other two are AVR family MPUs with an integrated USB controller.

3.2.1 AVR microprocessor + FTDI chips

The FTDI chips [6] offer a very easy-to-use solution. There are basically two black boxes on each side of the connection, connected by USB and offering fast UART to the outside.

One of the black boxes is a Windows driver that installs a new COM port; the other black box is the FTDI chip itself that converts the incoming USB into UART.

However, the existing UART communication libraries send the data in frames secured by CRC. Sending single bytes from these frames one at a time through USB would mean unnecessary overhead as each byte would be secured separately by the USB. On the other hand, the communication speed is not a very important issue for this project.

Pros:

- An instant solution with no work at all involved.

Cons:

- USB solution without any advantages of the real USB.
- Communication with an unnecessary overhead.
- UART on the microprocessor is still the limiting speed factor.

3.2.2 AT43USB355

The AT43USB355 [7] chip from Atmel is an all-in-one solution, an AVR family MPU with an on-chip USB available. At the time of the decision, the main problem was a quite unclear frequency of this chip. It requires a 6MHz crystal; the documentation claims it runs off a 12MHz clock generated by the USB hardware and admits the MPU can sometimes have single cycles deviating by up to ±20.8ns, which is ¼ of the normal 83.33ns clock cycle.

Assuming this could happen during the normal run of the MPU and wouldn't be a very rare occurrence, it might prove fatal to any precise time measurement. This worry has been dismissed by the project supervisor, who claimed Atmel would never market a chip with such a behavior occurring in any significant number of cycles. My opinion is that the matter should have been given more attention, because the precise time measurement was very important.

The unclear frequency aside, this chip seemed very good. Our functional code from the ATMega 128 would be easy to port. Atmel offered libraries supporting both Human Interface Device (HID) and "non-HID" (doesn't specify which exactly) classes and a Wizard [14] for generating the necessary code for the USB communication.

I also found an example for using a Communication Device Class (CDC) driver [8], which would offer functionality similar to that of FTDI chips in case we couldn't get the true USB communication working or were pressed by time. The advantage of this solution over the FTDI is that the hardware black box is integrated in the MPU itself, so there would be no UART bottle-neck as between the FTDI chip and MPU.

The main disadvantages are the lack of UART and a JTAG connector. The former would be useful for testing the final PCB, because we wouldn't have to be testing both a new communication and a new PCB at once. The later means there is no way to debug the MPU code directly in the MPU.

This chip costs only 120CZK (4-5USD), which is about a fifth of the cost of W3100A, the most interesting TCP/IP solution.

Pros:

- An all-in-one solution, which should help reducing possible errors in the PCB construction.
- Apparently very good support from Atmel.
- Possibility of an easy solution based on UART driver emulation

Cons:

- Unpredictably unstable clock can cause error in temperature measurement up to ±0.11°C cumulative for each instruction with a clock deviation.
- Doesn't have UART for simple testing or JTAG for debugging, so if USB turns out to be problematic it might be hard to get the MPU working.
- Writing USB driver will require Windows Driver Development Kit (DDK), which, unlike Software Development Kit (SDK), is not free.

3.2.3 AT76C713

The AT76C713 [9] MPU from Atmel is another all-in-one USB solution that integrates a USB controller on an AVR family chip. Unlike all the previous solutions, this chip runs at 48MHz, which means it is 4-6 times faster than the rest. The problem was that besides the quite good technical parameters there had been almost no other documentation when the decision was made.

Pros:

- High frequency leads to temperature measurement error as low as ±0.11°C
- All-in-one solution means quite simple PCB design.
- Contains UART that can be used in case of problems with USB

Cons:

- No documentation, no firmware guide, no mention of USB libraries.
- The only available examples are for USB-to-UART and USB-to-IrDA adapters.
- All the internet development boards contained only two references to the chip, both introducing it as a new addition to the Atmel product range.

3.3 Research summary

TCP/IP solutions mean very easy programming of the PC side and easy portability of the device to virtually any place that is accessible by any kind of Ethernet network the controlling computer is on, including Internet. The disadvantage of the software solution (3.1.1) is that it needs more MPU resources than we can spare. This mainly represents a high number of clock cycles during any time measurement, but quite a lot of available memory is consumed as well. The disadvantage of the hardware solution (3.1.2) is that despite the not-so-high price of the chip itself, it is still expensive compared to the USB solutions and the development would probably require the 300USD development board.

The USB solutions should prove much cheaper, and with an adequate support from the manufacturers should be equally easy to implement. The low portability of the device because of maximum cable length (5 meters) is not really an issue, because the

device will almost exclusively be in a test lab just a few meters away from the computer the tests are ran on.

Because the project supervisor felt the decision should be made quickly and we should move on to other tasks of the project, he decided to use the AT43USB355 (3.2.2) microprocessor, despite the possible issues with unstable clock period.

I am aware that this analysis doesn't cover many of the hardware issues, like memory sizes compared to the expected size of the program, number and size of available counters or number of free external interrupts available with each solution. The actual programming of the device was assigned to colleague and I assume he will describe this side of the problem in his bachelor thesis [4].

4. Analysis

This analysis focuses solely on the PC side of the project; for the tempLab device analysis please refer to my colleague's thesis [4].

The PC software had a very explicit specification including the required decomposition (see ***Figure 3***), the offered functions and the technologies to be used. I will therefore focus only on the algorithm analysis, as neither the technologies nor the decomposition should be changed.

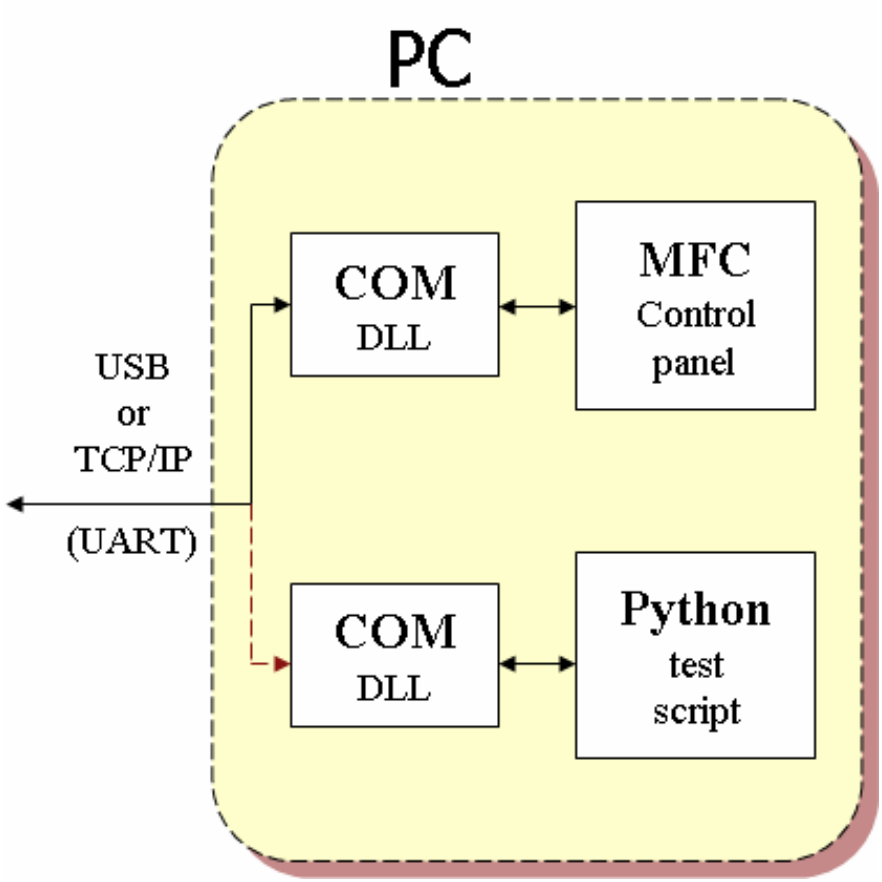

Figure 3 : PC decomposition

4.1 Communication module

The requested communication model was master-slave, therefore only the PC would ever initialize the communication, device will only respond to the received commands. This is a very simple model for both the device and the PC part, where device will cycle in an infinite loop waiting for the incoming commands, while PC will never have to listen for a device initialized communication.

Two main things had to be examined and analyzed in this module, first was the COM technology used for communication with both the GUI application and the Python scripts; the other was an option for calibration of thermometers.

4.1.1 Common Object Model

The Common Object Model (COM) [10] technology has one main goal: to allow an application written in any language use functions of programs and libraries written in any language, with virtually no limits of what languages are used.

To call an already compiled method, only two things need to be known. Where in the memory does it start and how are the parameters passed.

The first requirement is fulfilled using inheritance. If methods of a compiled object (be it a program or DLL) have been called directly, the caller program would have to be recompiled every time the object changed, because the relative addresses of methods change as well. Instead, we define a set of abstract classes (interfaces) with purely virtual methods. The actual object inherits from these, but the calls are made to the interfaces. That way adding or removing a single class variable won't require recompilation of all the programs calling its methods.

COM objects also use counted references to keep track of the number of existing pointers to them. The reason for this is that no application except for the object itself actually knows how many pointers to the object are there, so determining when the object can be safely deleted would be virtually impossible without the counted references.

The standard also defines an Interface Description Language (IDL) that is used for universal interface description, specifying methods and their parameters.

IDL supports many of the standard variable types known from C and C++, but also has some special types for passing strings, arrays and objects. But as the original specification of this project never assumed passing anything but numbers between the PC applications and the tempLab device, I have not studied this topic very much.

From the rough description of the method call mechanism, it is clear that a typical COM client program needs an a priori knowledge of the interface's definition at the development time. This is not a problem with any of the compiled languages, as the required information is supplied by either C++ style header files, or type libraries (Java and Visual Basic).

Another way had to be introduced for non-compiled languages like JavaScript or Python to perform static method calls. It is an IDispatch interface that allows script languages to call methods by their name, without any a priori knowledge of the method's address or even existence.

4.1.2 COM implementation

Because I had had no prior experience with this technology and was required to use Visual C++ 6.0 anyway, I decided to use the ATL wizard to generate the whole DLL and COM part of the tempLabCOM library, leaving to me only the method implementation itself.

The disadvantage of this approach is that it allows only one user interface besides the standard interfaces for error reporting and static method calls. This might pose a problem during testing of the tempLab device, as some methods should be available only for the testing and not for the normal usage. On the other hand, this will probably concern only the temperature methods, where the testing will require raw uncalibrated values.

The time spent modifying the code to hide a single method after all the testing had been done is incomparably shorter than learning and writing the whole program by hand, so the ATL was an obvious choice from the very start.

4.1.3 Thermometer calibration

Even though the manufacturer should provide the thermometers calibrated, it would be unwise to rely on that, so a calibration would probably be useful.

Our thermometers [18] have a higher total error (±0.7°C) than a non-linearity error (±0.2°C) so a simple offset calibration should be enough. But as other thermometers might be considered in the future, I have decided to provide more calibration options.

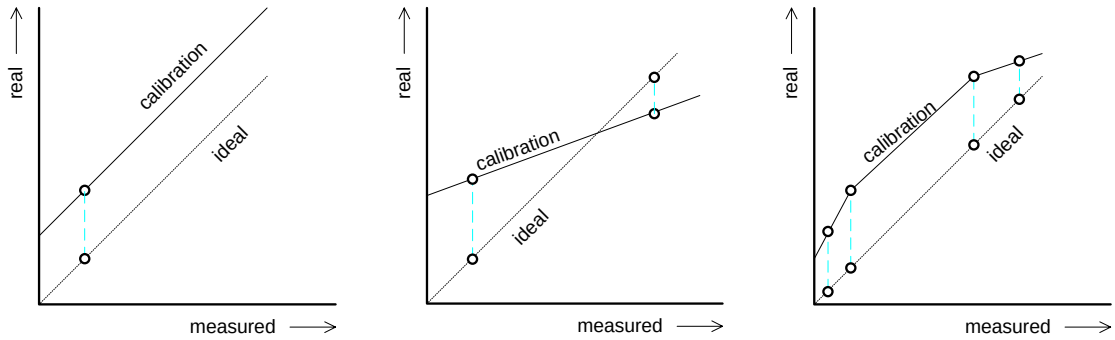

Figure 4 : Thermometer calibration a) Offset; b) Linear; c) Multi-line

- An offset calibration (***Figure 4***, a) – is the basic calibration which requires only one calibration point and simply adds the difference between the measured and the calibrated temperature to each measurement.
- A linear calibration (***Figure 4***, b) – allows to compensate a wrong slope of the DC-to-temperature function, for example when a thermometer measures correctly at 20°C, but gives 90°C instead of 100°C when measuring temperature of boiling water. Requires two calibration points.
- A multi-line calibration (***Figure 4***, c) – allows to compensate non-linearity of the conversion function.

It should be noted, that the calibration methods requiring more than one point should have these points set apart enough to prevent eventual troubles with a linear approximation and to actually have a positive effect.

To illustrate the later, let's assume that we make two calibrations set apart 1°C, for example at 20°C and 21°C. With our thermometer allowing the measurement error up to

±0.7°C, the worst case scenario is measuring 20.7°C and 20.3°C respectively, giving us actually a falling slope, which definitely is not the desired result.

It is obviously desired to have this calibration info transferable between computers (i.e., in a file) and accessible from both GUI and Python script (i.e., at a known location). For the file format I chose the well-known ini format that was used in pre-95 Windows instead of registers, because it is well readable from both within and without a program. The location could be hardcoded to a folder on a system drive, but I chose a more installation friendly option of having the file's location in registry.

This solution introduces necessity of an installation process, but as COM DLLs also require a registry entry this could not have been avoided anyway.

4.1.4 Thermometer error correction

It was required to have an error correction of the temperature measurement besides the calibration. Two options have been considered:

- Reporting too steep changes in the temperature as errors.
- Averaging several temperatures.

The temperature would be measured inside what is basically a miniaturized PC, so we could expect there would be no steep changes in the temperature. However, the first approach still requires some regular periodicity of the measurements, which is neither in the specification, nor can be expected from the known test patterns. Therefore I opted for the second approach, specifically measuring six values, dropping the highest and the lowest temperature and averaging the other four.

4.1.5 Request-response synchronization

The possibility of multiple applications trying to communicate with the tempLab device at the same time posed several levels of synchronization problems.

The lowest level is the UART communication, where only one process should be sending (or receiving) the outgoing (or incoming) frames at once. Otherwise the device would get two intermixed frames, respectively the applications would each get only part of an incoming frame, both resulting in failed CRC checks, timeouts and so on, depending on who got the stop byte.

The next level is quite similar. Let's suppose we protect one transaction with a mutex, so when an application is transmitting, all the other applications have to wait until it finishes. This poses a problem with master-slave communication, because there might be someone else sending a command between our command and response, so with sequence A writes, B writes, B reads (response for A), A reads (response to B) the whole communication would have to be repeated. The mutex should therefore obviously be set over the whole send-receive cycle.

The last group of problems is again based on the atomicity of commands. It was deemed necessary [4] to have very short (byte-wise) commands, therefore a sequence of commands for, for example, a thermometer measurement would be as follows: set the thermometer ID, start the measurement, check the state of the measurement in a loop till it is finished, retrieve the temperature.

It is obvious, that if some other process intervenes in this sequence, for example setting the ID to a different value before the measurement is started, then the whole measurement would be invalid and would have to be repeated.

There are three possible solutions to this problem. The most obvious and simplest is to let only one process be connected at any given time.

Another is to protect each one of the methods individually, thus preventing interfering communication when the application is actually communicating, while letting two different applications share the device without the necessity of connecting and disconnecting every time. This would allow the GUI application to check the status of the tempLab device when a Python controlled script was running.

Probably most useful would be a modification of this approach that would lock the "aggressive" commands like NetBreakers and turning the tested devices on and off to a single application, while allowing any application to check state of the device and measure temperatures. This would still allow the GUI application to check the state of a running test, while preventing the user from inadvertently interfering with the test.

For simplicity's sake I have chosen the first approach of simply locking the whole communication to the first application asking, but implementation of the third solution should be considered if and when the time permits.

4.1.6 UART communication

We have been provided with fully working libraries for the UART communication, including a suggested send and receive state machine and error handling. This was to be used as-is, as after porting the application to USB the libraries would probably be discarded anyway in favor of an USB specific communication.

4.2 GUI application

Once again both the language and the technology to be used were part of the specification, so the main part of the analysis was the application layout and management of the tempLabCOM method calls.

4.2.1 tempLabCOM method calls

With the ATL wizard already used to generate the testLabCOM DLL, I have decided to use all the available support Visual C++ offers for the cooperation with COM DLLs, mainly the smart pointers wrapping the calls necessary for counted

references. This means that with the exception of initialization and release of the library, all the methods could be simply directly called via a smart pointer.

Still, I have decided to wrap all the necessary calls in a wrapper object (CCOMWrapper). One of the reasons was to allow people who have never heard about the COM technology to use the library right away. The second reason was to handle correctly the case of failed initialization and prevent calls to methods of a library that has not yet been loaded into memory.

4.2.2 The layout

The purpose of the GUI application had been defined as testing the tempLab device functionality, not doing any real tests of the PROFINET devices. This means that while the application doesn't require any repetition or statistics of the used functions, it should offer an easy access to any and all the methods available from the testLabCOM DLL.

The dialog is required to have a reasonable size, to be viewable on notebooks with resolutions as low as 1024x768. This means there are basically just two possible layouts.

The first layout would be a very simple dialog box with a single drop-down list containing all the supported commands and an area where the corresponding editboxes and results would appear. The advantage of this approach is a really simple implementation. The disadvantage is also quite clear as seeing only the very last command leads to much confusion. It would probably result in a necessity of notes outside the program, to record the settings and then compare whether the results of are as expected or not. This idea has been dropped without anything but very rough sketches in favor of the following solution.

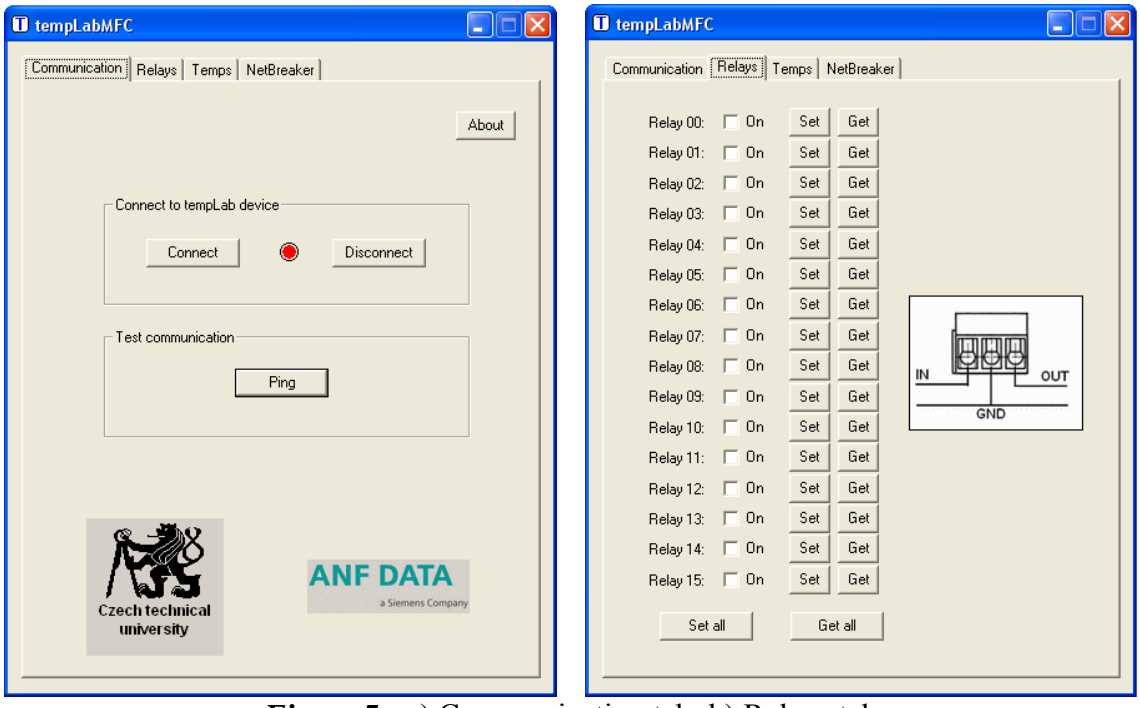

Figure 5 : a) Communication tab; b) Relays tab

The second possible solution is based on the fact, that the device's functionality can be divided into three distinct groups. This led me to an idea of a tabbed dialog with four tabs, three for each function group and the fourth for the methods dedicated to the connection between a PC and the device.

- Communication tab (***Figure 5***, a) – has a very simple design, as there are not many communication functions. The **Connect** and **Disconnect** buttons basically control the mutex discussed above (4.1.5). The red-or-green signal light between the buttons signalizes status of the connection; **Ping** is the simplest command for the device that only checks whether it is alive. One purely optical problem is the background of logos, which is based on inconsistency of dialog box color on different Windows versions.
- Relays tab (***Figure 5***, a) – is a bit more interesting. I've once again opted against any kind of a drop list for selection of the relay to control. It would offer very little optical comfort, especially considering the functions that set and get all relays at once. I think this layout is very self-explanatory, as each GUI should. The checkboxes tell whether each relay is on or off; **Set** and **Get** buttons set and get the value of their respective relay and the **Set all** and **Get all** buttons set and get values of all the relays at once. The rest of the space is filled with a diagram showing how exactly the relay connectors should be used, without a necessity of looking it up in the documentation.

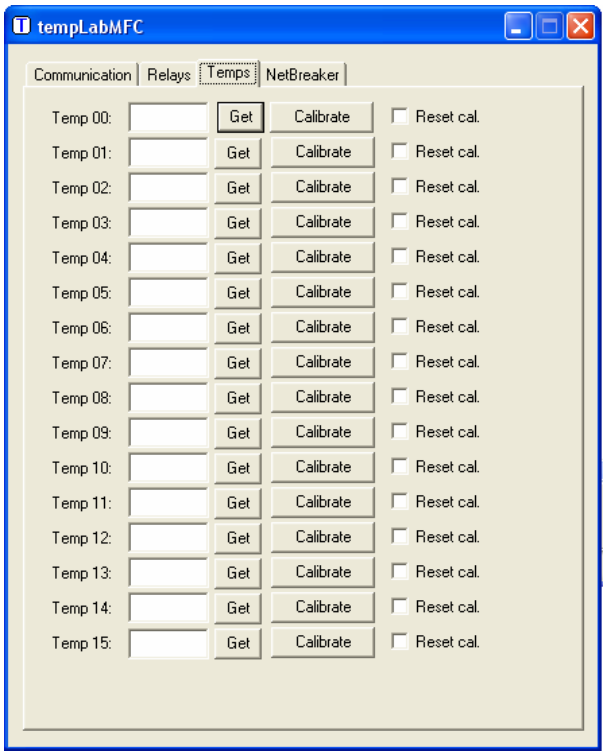

Figure 6 : Temps tab

- Temperatures tab (***Figure 6***) – will probably be the most used one, because besides the obvious function of checking the temperatures of each

thermometer it also offers a user-friendly way to calibrate them. Once again the GUI is pretty much self-explanatory. When a **Get** button is pressed, a measured (approximated and calibrated) temperature appears in the appropriate editbox. The thermometer calibration requires only entering the correct temperature into an editbox and pressing the **Calibrate** button. Normally this just adds another point to the calibration curve (4.1.3), but all the previous calibration info can be deleted simply by checking the **Reset** checkbox before clicking on **Calibrate**.

- NetBreaker tab has not been designed at this point, because it was still unclear what this function will (and can) do.

4.3 Python script support

Python is an object oriented non-compiled script language and, as discussed in 4.1.2, COM natively offers IDispatch interface for non-compiled languages. However, this is not the only available solution, as Python allows direct implementation of C++ libraries which was seen as an alternative, and probably faster, way to call COM methods.

Unfortunately, this way wasn't documented nearly as well as the IDispatch way. It was also considered, that the execution speed of the test itself probably wasn't critical, otherwise it would be written in something faster than a script language. Therefore it was decided to go the easy and well documented way of using IDispatch instead of spending days on saving small fraction of the script's execution time.

As for the end user access to the methods I again went for the wrapper way, thus allowing usage of the library with no knowledge about the COM technologies whatsoever.

5. Solution

For solution of the tempLab device please refer to my colleague's thesis [4] as I will focus here only on the solutions of the PC software.

5.1 Communication module

All the methods of COM interface have to be in one class, so I decided to at least group methods by their purpose in different files.

5.1.1 COM implementation

As stated in 4.1.2, it was decided to use ATL wizards for implementation of COM technology. This means that after the original class generation, the only steps necessary for adding a new method that should be available to the outside were filling its name and parameters in the wizard.

With parameters comes an interesting feature of this technology and that is error reporting. Passing the parameters back to the caller is done only by reference as the return value is fixed to an HRESULT type. HRESULT is a numerical type used in a way very similar to the standard C language error reporting via an integer return value. The advantage of HRESULT is that it has a unified format, specifying meaning of group of bits, allowing distinguishing between universal and application-specific errors. It is also possible to pair it with another of the generic interfaces, IErrorInfo. This interface allows us to set a text description of an error instead of just its numerical representation. Technologies chosen for implementing both C++ and Python wrappers interpret this as exception with all the error info readily available without any further processing, so this text description error reporting is used whenever an error occurs. For future translatability of the application, all the error texts are stored in a resource file instead of being hardcoded in any way.

5.1.2 Communication protocol

The protocol for communication between PC and the device have been designed by my colleague so I will only describe its basic principle.

It is a union type, with the first byte representing on opcode and the rest represented in various ways, based on the opcode value. Opcode's most significant bit represents direction of the command (PC-to-device or device-to-PC) while the other seven are the actual command.

This allows for very easy error checking, especially when combined with the device returning the sent data (thermometer ID, requested relay states, etc.) where necessary.

5.1.3 UART communication

As sated in 4.1.6, the UART communication has been provided as ready-to-use libraries and a state machine for sending and receiving commands.

The communication is implemented in a **SendRecBlocking** method which takes three parameters. Pointer to a structure representing command for the device, pointer to an identical structure where the response will be stored and parameter that allows to suppress throwing error messages.

This method, as the name suggests, sends a command to the device and waits for the response. It uses the methods offered by the protocol (5.1.2) and also by the lower layers to check if the received message is correct and when it is not it reads out the whole queue, repeats the command and after three failed attempts reports an error. Of course it can also time out in case the device is or gets disconnected.

5.1.4 Relays methods

The implementation of methods dealing with relays (i.e. with turning tested devices on and off) is pretty straightforward.

Each method starts with checking whether the DLL is or is not connected to the tempLab device. This is true for every single method in the interface. Then it checks for validity of the requested values (also true for any method used to set anything).

After this, it simply sends the request for either setting or getting relay states to the device. The only exception is **GetRelay** that has no directly corresponding command in the protocol so it calls **GetRelayStates** instead and extracts the single requested relay state.

5.1.5 Thermometer methods

The methods dealing with the temperature measurement can be roughly divided into three groups.

The first group consists of **GetOneTemp** and **GetRawTemp** methods.

- **GetOneTemp** – sets ID of the requested thermometer, starts the measurement and waits till it is finished. Then it fetches the measured time lengths (***Figure 7***) to compute a duty cycle and the corresponding temperature.

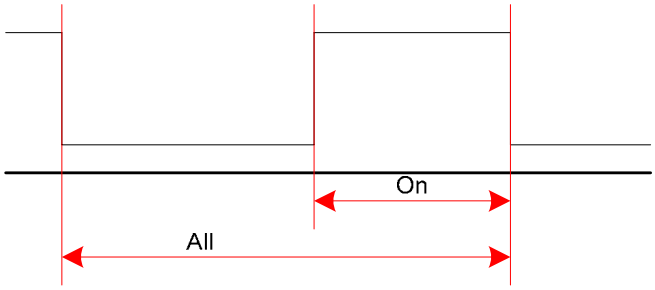

Figure 7 : PWM time values

- **GetRawTemp** – Calls **GetOneTemp** six times to get six measurements from the device, finds the minimal and the maximal value and returns the average of the other four as discussed in 4.1.4.

The second large group consists of methods dealing with the calibration. The calibration informations are stored in a linked list using a **t_Calibration** structure, which contains the temperature measured by the device, the real temperature and a pointer to the next member of the list. This list is kept sorted by the measured temperature.

The methods are:

- **LoadCalibrationInfo** – loads calibration from the ini file (4.1.3) into a structure consisting of sixteen linked list using **InsertCalibration**, and thus sorting the calibration in the process.
- **SaveCalibrationInfo** – saves the calibration info from the linked lists into the ini file and sorts the info in the process.
- **EmptyCalibration** – empties all the calibration linked lists.
- **InsertCalibration** – inserts new calibration info into the correct place in the calibration list.
- **GetClosestTemps** – finds the closest two entries in the calibration. This can be either one higher and one lower, or both higher or lower, which means that the measured temperature is lower respectively higher than any calibration point.
- **ThermCalibration** – Loads calibration info, measures the temperature (using **GetRawTemp**), assigns it the calibrated value and inserts the result into the calibration info which is then saved to disk again.

And finally **GetTemp** applies one of the calibration methods (4.1.3) based on the number of calibration points available to the temperature measured by **GetRawTemp**:

- No calibration available – returns the measured value without any further modifications
- One calibration point – returns the value modified by an offset.
- Two or more points – gets the two closest calibrated temperatures using the **GetClosestTemps** method and then applies a linear approximation to get the calibrated value.

5.1.6 NetBreaker methods

At this point NetBreakers haven't had a clear specification and it was obvious that there will have to be some serious changes to either the requirements or the hardware (see [4]), so I only implemented a method that sent requested values to the device, just to test the protocol.

5.1.7 MIDL compiler

Visual C++ 6.0 contains a program *midl.exe* for compiling MIDL (Microsoft Interface Definition Language) files, which are basically IDL files as described in (4.1.1). Unfortunately the one that's installed with Visual C++ 6.0 (including Service Pack 6) is corrupted and cannot compile IDL files created by the VC6 itself. It needs to be replaced by the *midl.exe* and *midlc.exe* files from Platform SDK.

5.2 GUI application

5.2.1 tempLabCOM method calls

I have encountered no troubles at all while programming the **CCOMWrapper** class. This was mainly because DLL calls had been very simplified by **#import** preprocessor command that generates all the necessary smart pointers and otherwise hides most of the implementation from a programmer.

5.2.2 Multitabs

To correctly implement multitabs I needed a modified dialog that would contain a pointer to **CCOMWrapper** used by the whole application. I also needed to override the automatic **CDialog** reaction to Esc (cancel) and Enter (ok), which would lead to the contents of a tab simply vanishing. That's because multitab in itself is just an empty rectangle with the well-known tabs at the top of it. Anything shown in the tab is a normal dialog inherited from the **CDialog** class.

This was done by inheriting a **CCOMDialog** class from **CDialog** and then changing the parent class of all the autogenerated dialogs from **CDialog** to **CCOMDialog**.

Another thing learnt from designing the multitabs is that the dialogs shouldn't be created and closed when tabs are changed, but instead they should all be created during the initialization (in the **PreSubclassWindow** method) and then only switched between using the method **ShowWindow**. This allows keeping values in the dialogs when switching between them and also has a lower overhead for switching.

5.2.3 System tray icon

Implementation of the request to minimize the application into system tray instead of the Windows taskbar consists of three simple steps:

- System tray icon – Creating the system tray icon is done by the **Shell_NotifyIcon** function which simply puts an icon into the system tray. The icon can be easily changed during the run of the application, which is used to indicate the status of the connection to the tempLab device.
- Minimizing – The dialog has to modify its behavior in response to the **ON_WM_SIZE** message. It has to hide itself instead of minimizing to the taskbar. The restore back action stays the same.

- Restoring – The minimized application lacks a clickable button on the taskbar, so another way to let it know it should restore its size has to be used. I registered a new event with the tray icon, which lets me know when there is an action performed and I restore the application when the action is double-click (i.e. when the user double-clicks on the tray icon).

5.2.4 Communication Dialog

This dialog is very simple, only showing buttons for connect, disconnect and ping functions, a traffic-light icon showing the connection status and the about button.

As all the buttons directly call the corresponding wrapper methods there haven't been any real development besides the layout design discussed in 4.2.2.

5.2.5 Relays Dialog

This dialog introduces one of the possible approaches for handling multiple controls (buttons, editboxes etc.) with similar function. In this case we have a group of sixteen checkboxes and then two groups of sixteen buttons each, the **Get** and **Set** buttons.

The naïve approach for handling all the button events would be to create a handling method for every single button and hardcode it to be tied to its respective checkbox. This would mean 2x 16 almost identical methods, which by the definition of functional programming is definitely a wrong approach.

Therefore instead of an **ON_BN_CLICKED** macro that assigns a handling method to a single control and action, I have used an **ON_CONTROL_RANGE** macro that assigns handling method to an action and a range of control IDs. This requires some simple manual modification of the *resource.h* file to have all the IDs ordered with the next one being always one higher than the previous.

Once all the checkboxes, **Get** buttons and **Set** buttons are ordered, each button's respective checkbox can be easily accessed, because the numerical difference between IDs of a checkbox and its respective button stays the same for all the buttons of the same type.

This means that instead of 2x 16 methods I implemented only two, one for the **Get** and one for the **Set** buttons. The methods get an ID of the clicked control, use the known checkbox-to-button difference to find the appropriate checkbox and either set it according to the value read from the tempLab device, or set relays in the device according to the checkbox's status.

The **Get all** and **Set all** buttons then simply use a **for** cycle to set (respectively get) state of all the checkboxes.

5.2.6 Temps Dialog

The name of the thermometer dialogue have been set to "Temps" mainly to have a similar length of all the tab names and "Probes" wasn't considered intuitive enough.

Otherwise, the implementation is almost identical to the implementation of the Relays dialog.

Get button calls the wrapper's **GetTemp** method and puts the result into a corresponding editbox.

Calibrate button first checks if the corresponding **Reset cal** checkbox is checked and if so it calls the **EmptyCalibration** method. Then it uses the value in its editbox to call the **ThermCalibration** method, thus inserting another calibration point for a given thermometer.

It is assumed that the thermometer calibration will be done almost exclusively in this dialog, while all the other functions serve mainly to test the tempLab device.

5.2.7 NetBreaker Dialog

For the reasons given in 5.1.6 there had been no dialog made at this point and the whole issue will be dealt with separately later in this document.

5.3 Python script support

Implementing a Python COM wrapper is almost as simple, if not simpler, as implementing the wrapper in C++. All the methods check whether the COM DLL has been successfully loaded in the object initialization. If it was it directly calls the appropriate COM method, if it was not it throws an exception.

The only differences from this template are the **SetRelayStates** and **GetRelayStates** methods. Those have to convert between an array of true/false Python representation of the relay states and the bit representation (16bit number) of the same that is used as parameter of the COM DLL methods. This is because passing an array over the COM interface is a bit more complicated than simply encoding 16 true/false values into a 16bit number.

6. Detailed technical specification II & III

After several meetings with the testers, the NetBreaker specification had been redefined. As a result we had to add a CPLD to be able to perform all the required functionality.

6.1 NetBreaker

As was already mentioned several times, we have encountered some serious problems with specification and implementation of NetBreaker. For detailed analysis please see [4].

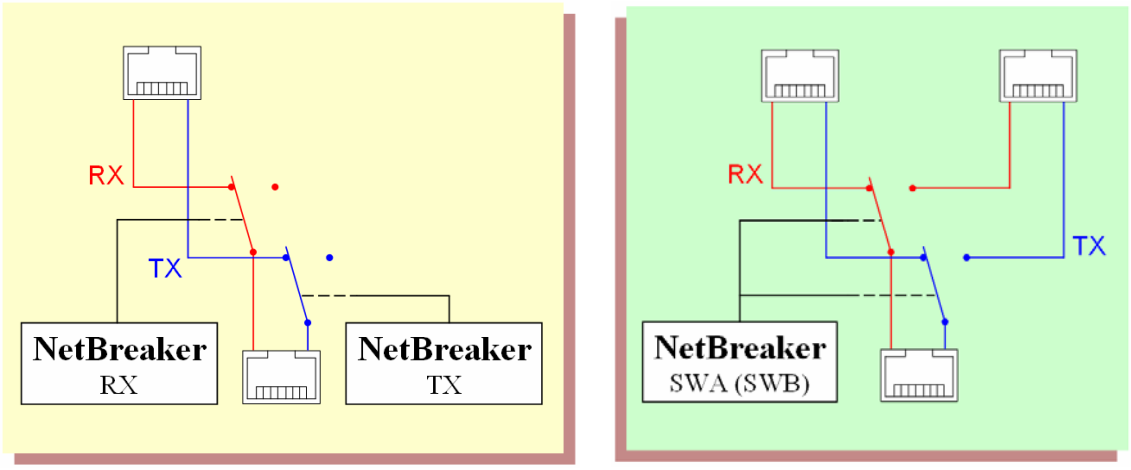

Figure 8 : NetBreaker functionality a) RX and TX NetBreaking; b) Branch switch

After several meetings the testers came up with new requested functionality:

- Allow absolutely asynchronous work of all four NetBreakers.
- One pair of In and Out RJ-45 connectors with RX and TX broken separately (***Figure 8***, a).
- Implement network branch switches SWA and SWB (***Figure 8***, b).

The last request was based on the fact that the dedicated LAN switch chip my colleague decided to use for NetBreaking offered an option to switch an input between two outputs. Testers decided that this would be useful and offer new testing options.

Another request that came several months after the first three was to implement several modes of NetBreaking. As will be shown later, the modes implementation introduced only minimal changes so I decided to include them in this specification.

The requested modes (***Figure 9***):

- CSS – Command, Sync and a T1, T2, T3 sequence ("Sequence") is the original NetBreaking sequence as described in chapter 2.
- ESS – External, Sync, Sequence will start the whole process by an external signal instead of a command from the PC.

- CS – Command, Sequence will ignore Sync, therefore ignore the T1 value (***Figure 2***) and start counting T2 immediately.
- ES – External, Sequence is almost identical to CS, except it is again started by the external signal instead of just a command from the PC.
- Ext. trig. – is a mode that completely ignores any times and directly connects the external signal to the LAN switch chip.

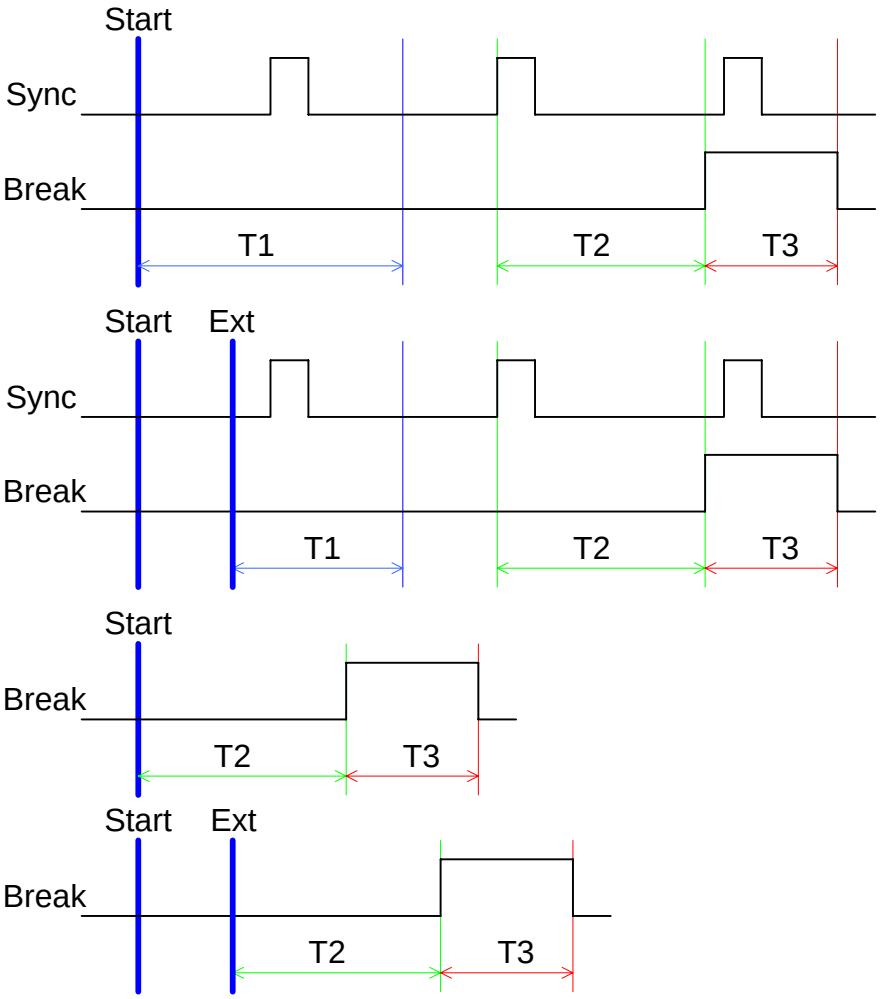

Figure 9 : NetBreaker modes a) CSS, b) ESS, c) CS, d) ES

6.2 CPLD

As shown in my colleague's thesis [4], this functionality could no longer be implemented in a MPU. It was decided to go for a programmable logic chip, because hardware implementation of the requested level of parallelism is very easy.

It was requested to fit the design into a CPLD, if at all possible, to reduce the number of chips on our PCB, because an FPGA would require an external memory chip.

Another two requests were raised after I turned in the first VHDL designs:

- All but the bottom level designs should be in graphics instead of VHDL to make the whole design more readable.
- Find and use a suitable naming convention similar to the Hungarian notation that is used in C programming.

7. Analysis II & III

It was decided to move all the functionality into the CPLD instead of just the NetBreakers, therefore solving the timing issue of the chosen MPU (3.3). The MPU would now be acting merely as an interface between the PC and the CPLD, which set us back a bit as most of the functionality was already implemented as a MPU program.

To choose a specific CPLD it was necessary to have an almost complete hardware design so we could correctly estimate the required size of the CPLD.

A new chip would also require new PCBs, so while my colleague focused on those I was assigned the development of the VHDL code itself.

7.1 tempLab device analysis

The device analysis included choosing a target CPLD based on some preliminary test designs and estimations and the actual analysis of the- VHDL design.

7.1.1 CPLD chip

After some preliminary estimations, test designs and synthesis attempts it was decided that the largest CPLD available from Xilinx (512 macrocells, therefore 512 DFFs) was not large enough. Even though it was possible the design would fit in, we would start at about 90% of the CPLD with little-to-none space for additional functions and design corrections.

Here I would have opted for an FPGA by the same company, because I had previous experience with the chip and the design tools from school. The project supervisor had objections not only because it would require another memory chip, but also because FPGA should be more sensitive to wrong voltage and the design already had some issues with 5V and 3.3V parts.

Therefore we went for an Altera Max II CPLD, which offered a single-chip solution with 1270 Logic Elements (***Figure 10***). From what I have learnt later on, my personal opinion is that this is an FPGA-structured chip with an on-chip Flash memory allowing a single-chip solution. If this was so, then the second reason for selecting CPLD over FPGA would not be satisfied.

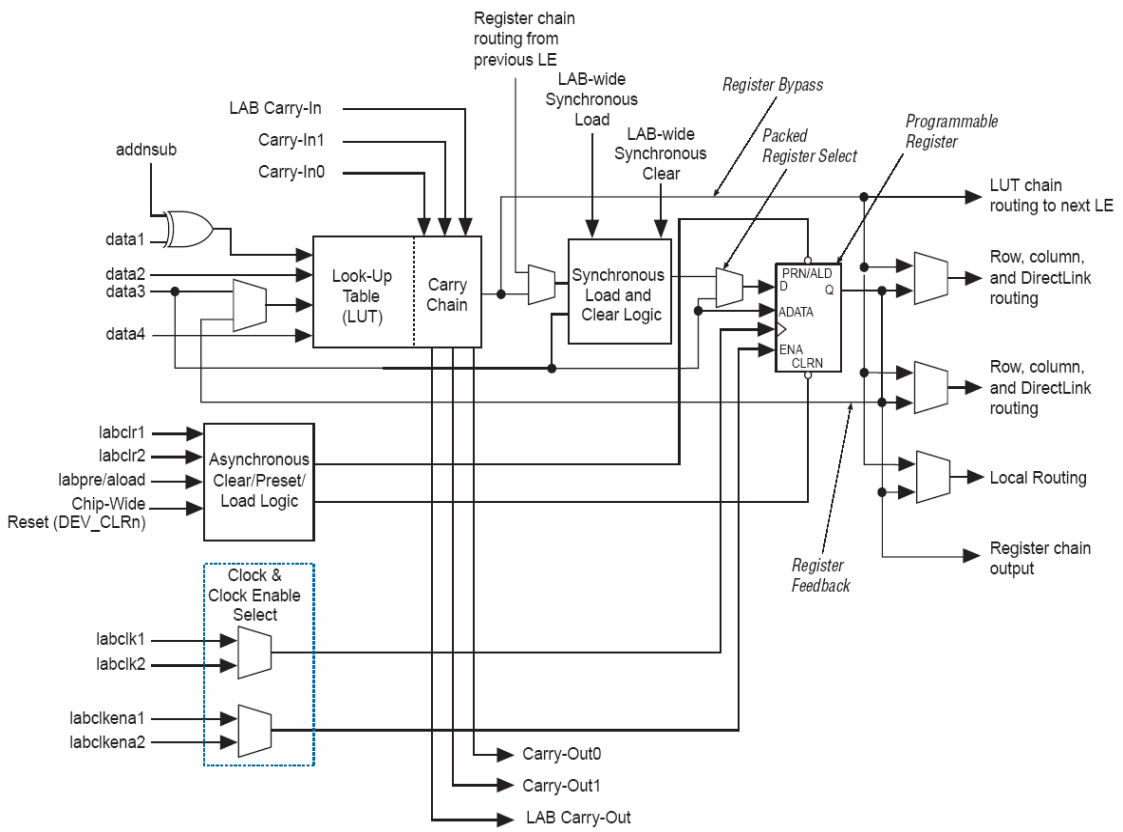

Figure 10 : Max II – Logic Cell

7.1.2 VHDL design

There had been several requirements set for the VHDL design from the start, but some others appeared during the analysis and will be mentioned in the respective chapters.

7.1.2.1 Requirement summary

For CPLD the following requirements had been set:

- A well-defined naming convention.
- All but the lowest level of design in schematics instead of VHDL.
- Fast and reliable communication with the MPU.
- Four independent NetBreakers that would allow both simultaneous and separate start.
- Relays control register.
- Temperature measurement.
- A high enough frequency to ensure a high enough precision of the time measurement.

7.1.2.2 Naming convention

I have found only one naming convention [11] and I have decided to use it. It appears that neither of the major CPLD manufacturers (Xilinx, Altera) uses any kind of naming convention in their examples and sample codes.

7.1.2.3 Design decomposition

It was obvious and also required that the CPLD implementation had a block structure (***Figure 11***), as the **BIU** (Basic Input Unit), **NetBreakers**, **Relays** and **Therm** (Thermometer) blocks were almost independent on each other.

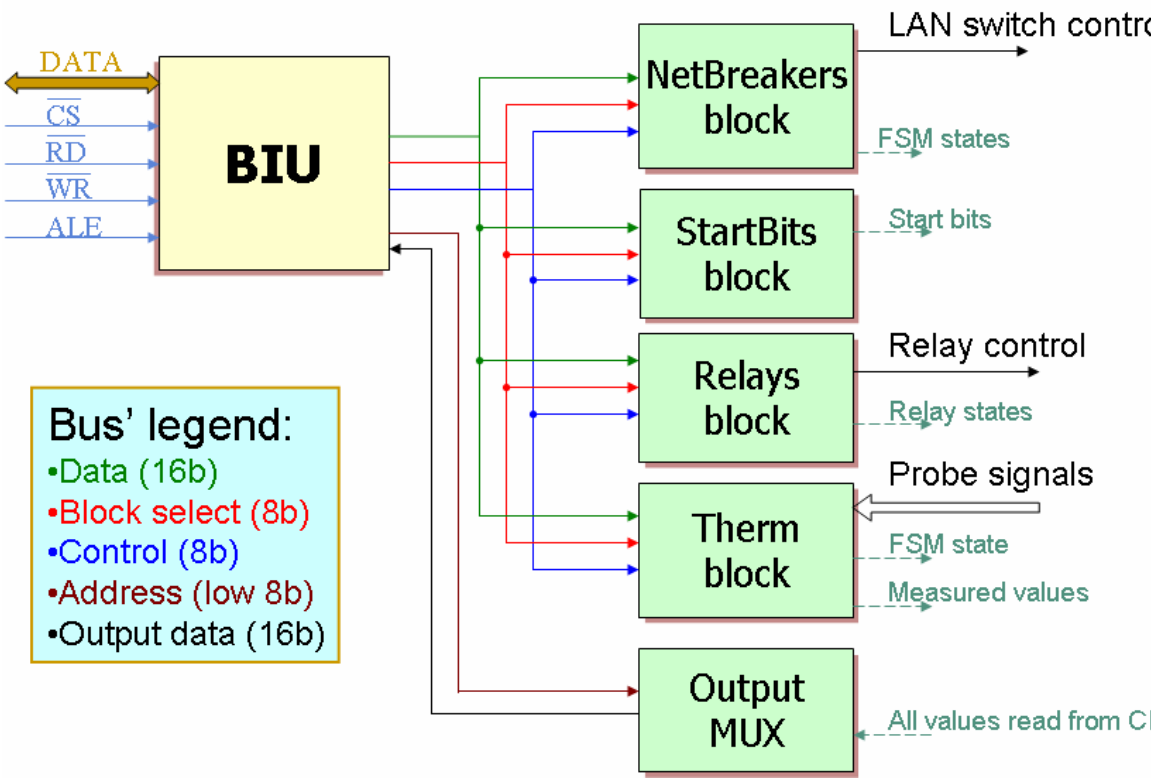

Figure 11 : CPLD decompositon

The **StartBits** block is a separate register that controls starting of all the finite state machines (FSM) in the design. It has its bits separately resetable to prevent an unrequired self-restarting when a FSM finishes.

The **OutputMUX** block is an alternative to a bidirectional inner bus, which has a very low support in the Max II CPLD. As the only values I need to read from the CPLD are the whole- and up-time of the PWM measurement in the **Therm** block, the relay states from the **Relays** block and the FSM states of the **NetBreaker** and **Therm** block state machines, a simple 4-to-1 multiplexer and a second unidirectional bus for the output values proved to be an adequate solution.

7.1.2.4 Inter-block communication

The communication between the blocks consists of several groups of buses and signals:

- Main buses:
  - **DataBusxD** – transports 16b data to all the blocks.
  - **BlockSelectxD** – is a one-hot encoded part of the requested address (for detailed description see 8.1.2) signal that activates the block we want to control. Four are for the **NetBreakers** block (one for each **NetBreaker**), one is for the **Therm** block and one for both **Relays** and **StartBits** blocks together.
  - **ControlBusxD** – is another one-hot encoded part of the requested address signal, this time it tells the selected block what it should do. It is specific for each unit and will be described in the appropriate chapters
- Output buses:
  - **AddressxD** – this leads the requested address directly into the **OutputMUX** block where it is used to choose the correct data.
  - **OutputxD** – leads the correct output data from **OutputMUX** into **BIU**.
- Other signals:
  - **StartxS** – signals start their respective FSM.
  - **ResetxR** – signals are used by the FSM to reset their start bit when already started to prevent the unrequired restarting (not shown in the diagram).
  - **TempReadyxD** and **NBReadyxD** – represent the FSM states signals in the diagram. They are combined into a single 5bit signal and then used as one of the possible choices for **OutputMUX**.
  - **PeriodTimexD**, **HighTimexD** (Measured values) and **RelayStatesxD** (Relay states) – are the other three possible choices.

7.1.2.5 CPLD-AVR communication protocol

We needed to design and implement a communication protocol (sometimes referred as a "bus cycle") between CPLD and AVR. The oddity of this communication is that while most protocols assume a fast CPU and a slow periphery, our CPLD is four times faster than the MPU.

The possible protocol families were:

- Two-way handshake
- One-way with sufficiently long control pulses

If the decision was solely up to me, I would have chosen a two-way handshake variant (actually had one implemented) because it is reliable and the code is easy to transfer between various platforms.

However I had been overruled on this and we went for the one-way protocol, because it has a lower communication overhead. I think this hurts the idea of my colleague's modular hardware solution, because the communication needs to be modified and tested with every new MPU.

7.1.2.6 Signal filtering

With the request for filtering of several input signals, namely Sync, external trigger and thermometer signals, an adequate filtering method had to be found.

Two possible and easy to implement solutions are:

- Majority from three samples.
- Two consecutive signals to change the output.

I have chosen the second option, because it requires one less DFF which had been considered the more critical resource because of the CPLD structure.

The method is very simple, the filter remembers its current output, the last sampled value and it takes another input of the same value to change the output value. The downside is that the output signal is delayed by two clock cycles.

7.2 Software analysis

The re-specified requirements mean almost no change to the software. The only things that have to be added are methods for sending all the necessary NetBreaker data to the device and a NetBreaker tab in the GUI application.

7.2.1 Communication

Because the CPLD implementation showed it would be easier to count NetBreaker times at the CPLD's native frequency instead of the required precisions it became necessary to convert the requested times to number of the CPLD clock cycles. As a side effect, this allows NetBreaking with a much higher precision than was originally requested. I trust we can easily claim precision ±2T, where T is a CPLD clock period, without any further mathematical proof.

To do this a new protocol opcode had to be introduced to allow getting the CPLD frequency from the tempLab device.

Another problem appeared with the Sync filtering. It takes the device 3.5-4.5T to recognize Sync, so extremely low times T2 (***Figure 2***) are not possible. Because this value doesn't take into account delays on input and output buffers, we did some basic measurements and found that with 48 MHz crystal, the delay between Sync and the NetBreaking with T2 = 0 is about 130 ns.

The **SetNetBreaker** method therefore subtracts this value from any requested T2 time and refuses T2 shorter than 130 ns. It also checks the requested times against values that can actually fit into the CPLD counters.

The filtering delay on the external trigger is insignificant, because it is added to the delay between the PC and the CPLD that is far greater.

7.2.2 GUI application

GUI had once again been a decision between an all-visible-at-once approach and choosing and performing single commands separately and sequentially. I have picked the all-at-once layout (***Figure 12***) for exactly the same reasons as before. It offers more comfortable and user-friendly look. Also allows results to be easily checked against the entered values without a necessity of notes outside the application.

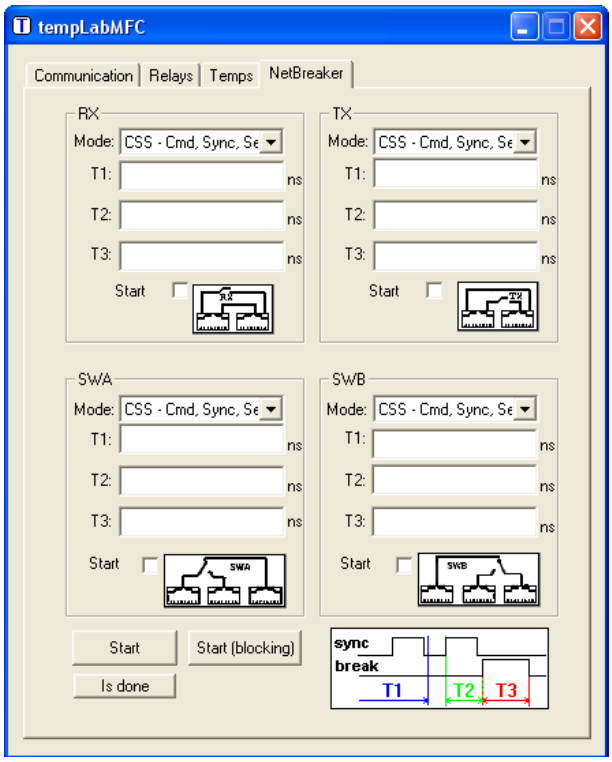

Figure 12 : NetBreaker layout

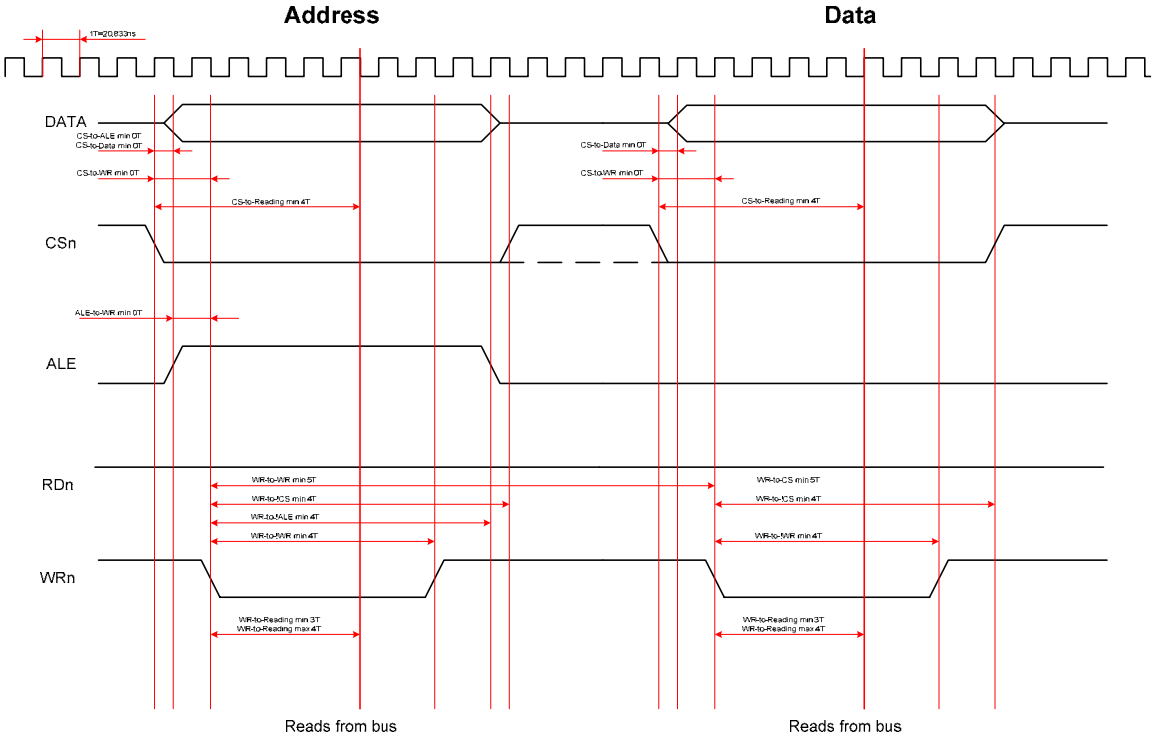

Figure 13 : MPU to CPLD transfer

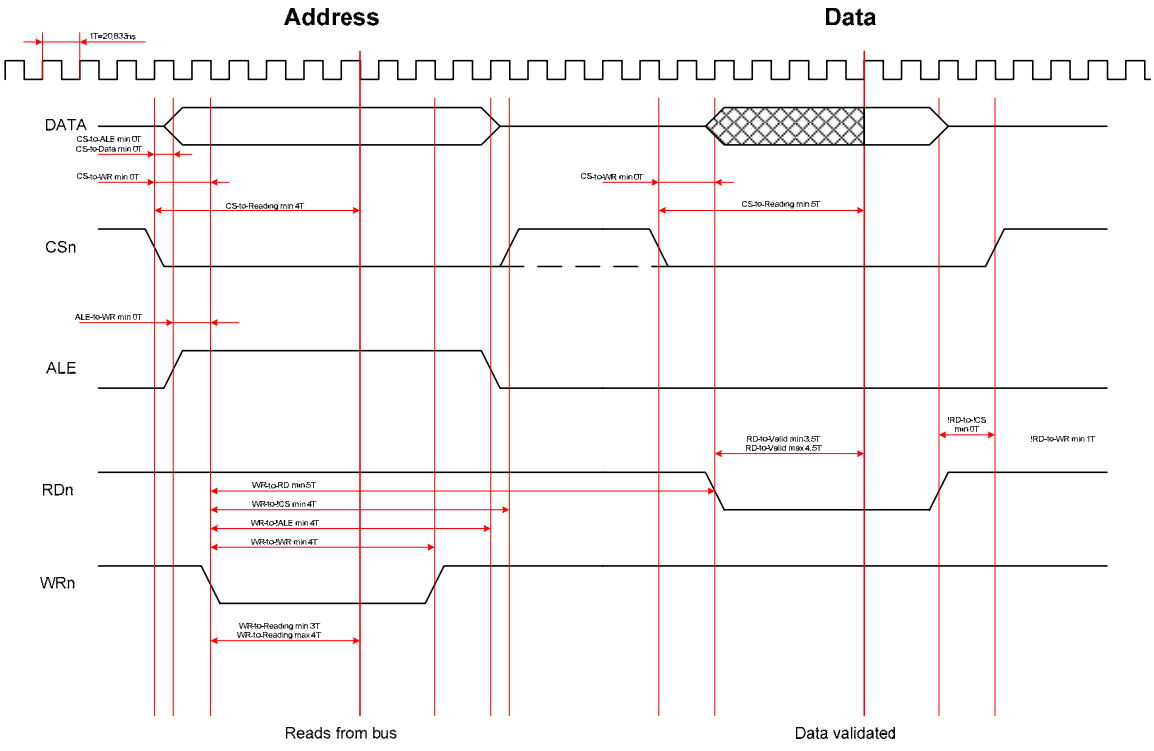

Figure 14 : CPLD to MPU transfer

8. Solution II & III

Solution will be also divided into two chapters, one dealing with the VHDL design of CPLD in the tempLab device, the other with PC software.

8.1 VHDL design

Here I will describe the chosen solution for communication between a CPLD and a MPU as well as function and structure of each of the VHDL design blocks.

8.1.1 CPLD-MPU communication protocol

It was required to implement some kind of glitch prevention here. To achieve this it takes two samples of each signal for it to be recognized as active. I have included description of writing a command into CPLD (***Figure 13***) and reading data from CPLD (***Figure 14***), for the complete sequences please see appendix A.

All the required times are defined relative to T, where T is a period of a single CPLD clock cycle. Currently we are using 48MHz crystal, therefore T = 20.833 ns. The uncertainty of some times (min and max) value is caused by the asynchronicity of MPU and CPLD clocks. The minimal time represents the situation when a signal in question changes just before the CPLD clock edge, the other extreme is when it changes just after the edge and it takes almost whole T period before it is sampled.

Half-periods in reading are caused by the behavior of an output latch, where the output data are sampled at the rising edge, but available at the falling edge.

For the full list available addresses and the corresponding meaning of data please see the appendix B.

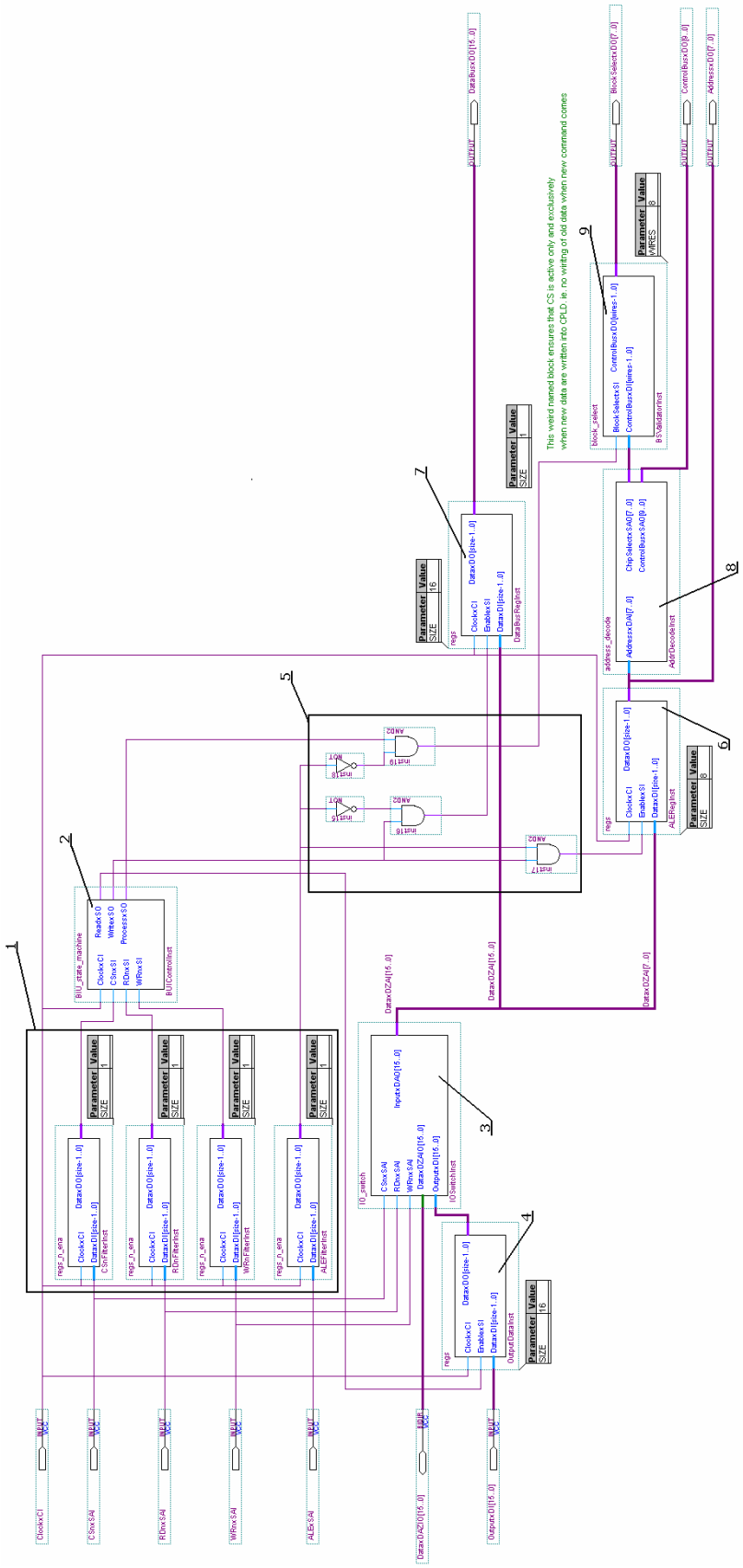

Figure 15 : BIU block diagram

8.1.2 Basic Input Unit

Implementing the protocol described in the previous section is the Basic Input Unit (BIU) with its block diagram shown on ***Figure 15***.

As can be seen, I have divided the **BIU** into several components (I will avoid the word "blocks" as it is associated with function blocks of the whole CPLD design):

- Control signals (**CSxSAI**, **WRnxSAI**, **RDnxSAI** and **ALExSAI**) are gated by DFFs (1), here represented by four 1bit registers (without clock enable).
- **BIUControl** (2) is a FSM that controls the whole **BIU** block and will be described in greater detail later on.
- **IOSwitch** (3) deals with the bi-directional **DataxDAZIO** bus and switches between:
  - Letting the **OutputxDI** port data (when CS and RD signals are active and WR is not) to **DataxDAZIO** bus.
  - Letting the **DataxDAZIO** bus signals onto the **InputxDAO** port (any other combination of signals)
- **IOSwitch** reacts directly to asynchronous signals (i.e. before they are gated) to minimize the risk of two devices transmitting on the **DataxDAZIO** bus.
- **OutputData** (4) is a register that latches the output data so they won't change during reading from the CPLD.
- The block of logic without description (5) is there to decide (based on **ALExSAI**) whether the data from **DataxDAZIO** bus should be interpreted as an address or data.
- **ALEReg** (6) latches addresses from the **DataxDAZIO** bus.
- **DataBusReg** (7) does the same for the data signals. From guidelines [12] acquired very late in the project it appears that this register is not absolutely necessary. I have decided to keep this register mainly because it has already been extensively and successfully tested in this configuration.
- **AddrDecode** (8) decodes a latched address onto two one-hot encoded buses:
  - The higher four bits of each address are decoded into a **BlockSelectxDO** signal, selecting which of the functional blocks is being addressed
  - The lower four bits are decoded into **ControlBusxDO** signal, signaling which of the select block's functions is required. (7.1.2.4)
- **BSValidator** (9) ensures that the **BlockSelectxDO** signal will be active only when the correct data are ready, because the **ProcessxSO** signal comes exactly one cycle after the **WritexSO** signal.

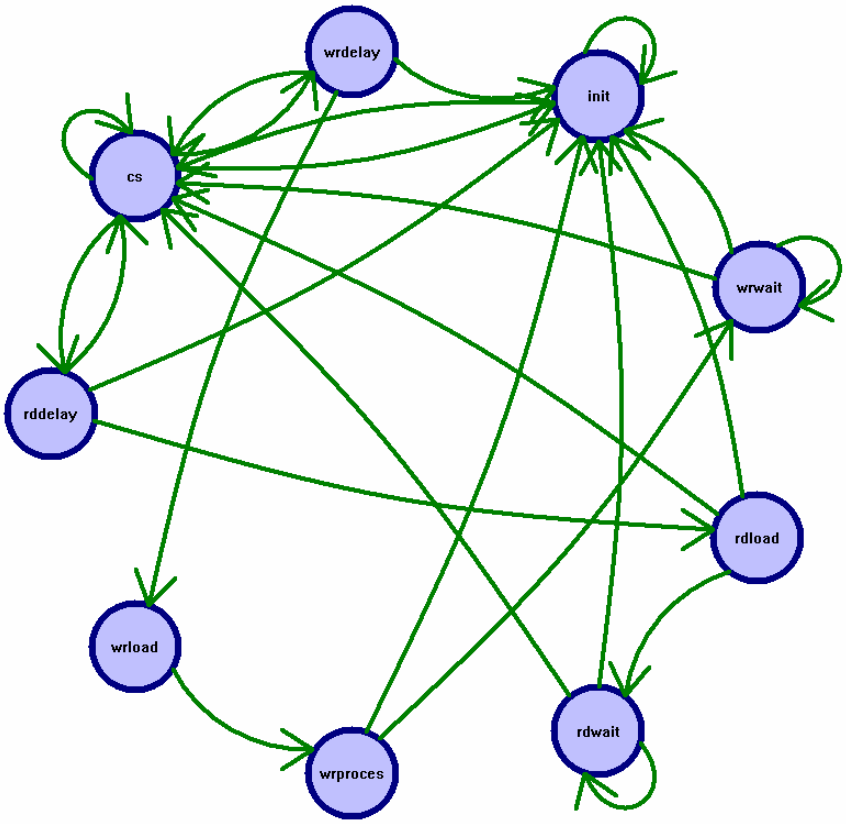

Figure 16 : BIU finite state machine

8.1.2.1 BIU finite state machine

To describe the BIU finite state machine I will use a diagram generated by Synplify Pro from the original VHDL code.

The first important thing that might be noticed about this FSM is the lack of a reset signal. This was part of the requirement for no external resets which turned up during the design process. The CPLD should reset all it's Logic Elements after power-on [13]. The latest tests show that this obviously isn't always so, but it doesn't really pose any problems for this FSM as whenever the **CSxS** signal is inactive the next state is automatically the **Init** state.

The states and possible transitions between them are:

- **Init** state – As already stated the FSM goes into this state whenever the **CSxS** signal becomes inactive. This transition has the highest priority and always overrules any other transition described in the other states. The only possible transition from this state is a transition into the **CS** state when the **CSxS** signal is active.
- **CS** state – This state ensures that the latched **CSxS** signal wasn't only a glitch, therefore conforming to the two samples requirement (8.1.1). When **WRxS** and **RDxS** are both either active or inactive, the FSM stays in this

state. It also returns into this state whenever such a combination of **WRxS** and **RDxS** signals occurs.

- **RDDelay** and **WRDelay** states – Another set of states to ensure that the sampled active **RDxS** respectively **WRxS** signals were not just glitches and there are two samples before it is acted upon. The transition is quite obvious, if the signal is still active with the next clock it goes into the respective **Load** state, otherwise it goes back into the **CS** state.
- **RDLoad** state – Generates **ReadxS**, a write enable signal for **OutputData**, to latch the data ready to be read from the CPLD (***Figure 15***). The FSM can go to either **RDWait** (if **RDxS** signal is still active) or back to the **CS** state.
- **WRLoad** state – Generates **WritexS**, a write enable signal that, depending on **ALExS** signal loads the **DataxDAZIO** bus signals into either **ALEReg** or **DataBusReg**. (***Figure 15***). This state is always followed by the **WRProcess** state.
- **WRProcess** state – This state ensures that appropriate **BlockSelectxDO** will be generated exactly one cycle after latching data from the **DataxDAZIO** bus.
- From both **Load** states, the FSM can go into the respective wait (**WRWait** and **RDWait**) states, where it waits until the **RDxS** (respectively **WRxS**) signal becomes inactive.

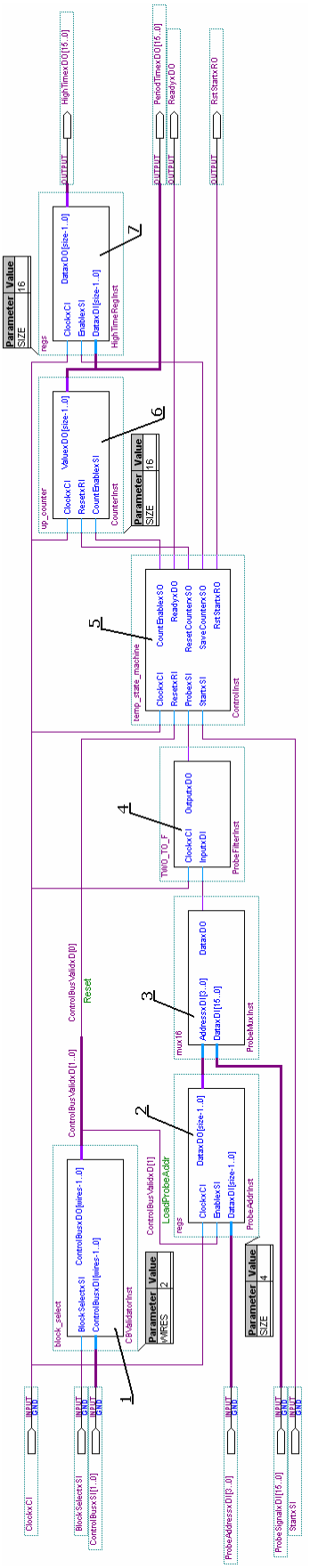

Figure 17 : Thermometers block diagram

8.1.3 Thermometers block

The block implementing thermometer functionality also consists of several components:

- **CBValidator** (1) performs logical AND between the **BlockSelectxSI** signal and the **ControlBusxSI** signals so the block will accept control signals only when it is selected.
- **ProbeAddr** (2) stores an ID of the measured thermometer.
- **ProbeMux** (3) then uses this ID to select a signal from the correct thermometer. The result is that we need only one set of counters and FSM for all sixteen thermometers.
- **ProbeFilter** (4) is there to gate the probe signal and prevent inaccuracies caused by glitches. It implements filtering method described in 7.1.2.6. The two clock cycle delay doesn't influence our measurement as it is the same on both a rising and a falling edge and we are measuring the time difference between those two.
- **Counter** (6) is used to measure a length of the pulse
- **HighTimeReg** (7) stores the length for which the pulse had been in logical high state. Sixteen bits is enough to measure PWM on frequencies as low as 730Hz (4), while the used thermometer ensures range 1-4 kHz.

$$f_{\min} = \frac{f_{clock}}{2^{bits}} = \frac{48MHz}{2^{16}} = 732.4Hz \quad (4)$$

- Both **Counter** and **HighTimeReg** are controlled by the thermometer FSM, **Control** (5)

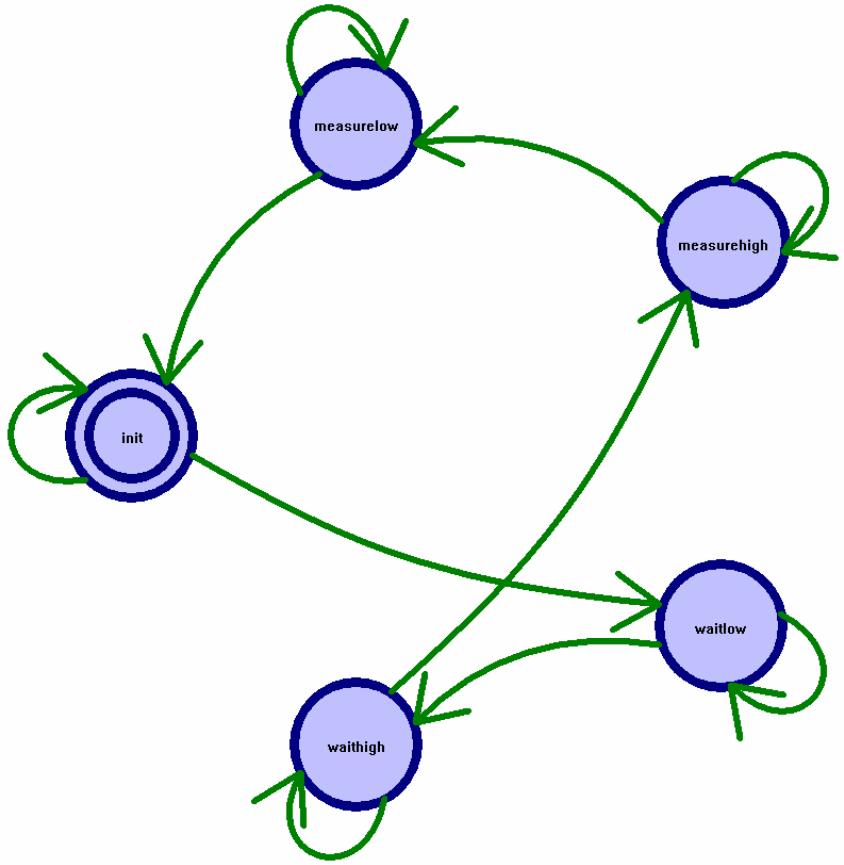

Figure 18 : Thermometer finite state machine

8.1.3.1 Thermometer finite state machine

The thermometer FSM is a bit simpler than the BIU one. This one also contains a reset signal which resets the machine into an **Init** state.

- **Init** state – The default state where the only active output signal is the **ReadyxDO** signal. When the **StartxSI** signal comes, the machine goes into the **WaitLow** state.
- **WaitLow** state – Both wait states are there to ensure, that measurement will start at the rising edge of the input signal. Both also keep resetting the **Counter**, and the Thermometer start bit. This state, as the name suggests, waits for the **ProbexSI** signal to go into low and then the FSM transits into the **WaitHigh** state.
- **WaitHigh** state – Here the FSM waits for the **ProbexSI** signal to go high. Then it goes into the **MeasureHigh** state where the actual measurement starts.
- **MeasureHigh** state – The FSM stays in this state until the **ProbexSI** signal goes low again. While in this state, both **Counter**'s **CountEnablexSO** and **HighTimeReg**'s **SaveCounterxSO** signals are active.

- **MeasureLow** state – The final stage of the PWM measurement. When the **ProbexSI** signal goes high again the machine goes back into the **Init** state. In this state only **CountEnablexSO** is active.

The result of this is that at the end of the measurement cycle the **Counter** contains the length of the whole PWM period, while the **HighTimeReg** register contains the length for which it had been in high state.

8.1.4 Relays, StartBits and OutputMUX

These three blocks are very simple and neither requires any complicated description:

- **Relays** block is simply 16bit register with each bit tied to one relay.
- **StartBits** block is a 5bit register with five distinct **ResetxR** signals, one for each start bit.
- **OutputMUX** is a 16bit multiplexer that chooses one of the four possible outputs date, based on the address in **ALEReg**. The four possibilities are the relay states, the period and the high time from the **Therm** block and the combined **Ready** signals from the **Therm** and **NetBreakers** blocks.

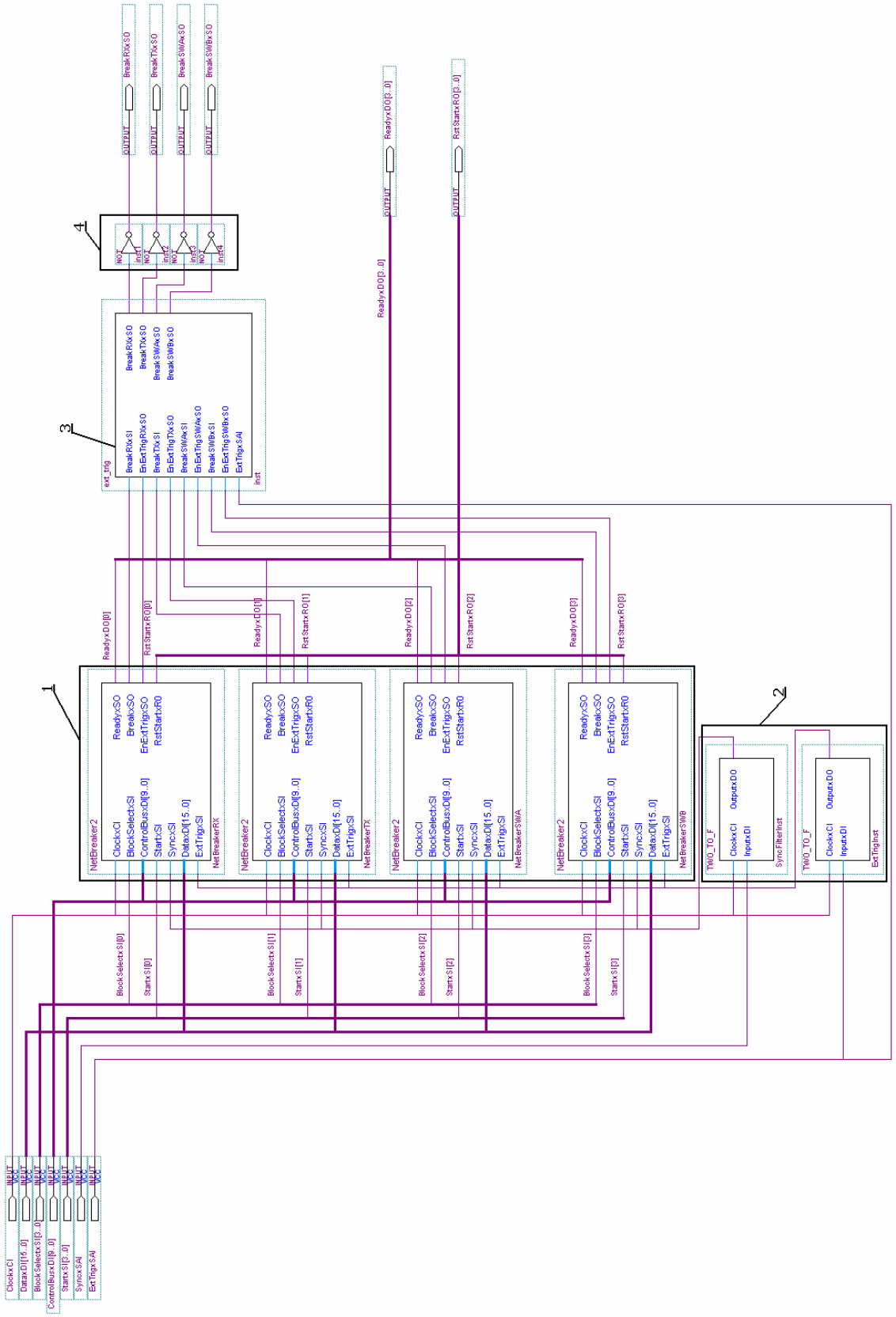

Figure 19 : NetBreakers block

8.1.5 NetBreakers block

NetBreakers block (***Figure 19***) contains:

- The **NetBreakers** themselves (1).
- Filters for both the **SyncxSAI** (2) and the **ExtTrigxSAI** (3) signal, again implementing the filtering method described in 7.1.2.6. This time the delay caused by the filtering is significant as discussed in 7.2.1.
- Logic for the direct external triggering (3) basically allows the **ExtTrigxSAI** signal directly onto the break outputs of NetBreakers in the Ext. trig. mode.
- Set of invertors (4) is there because the dedicated LAN Switches are active in log 0, while the control signals are active in log 1.

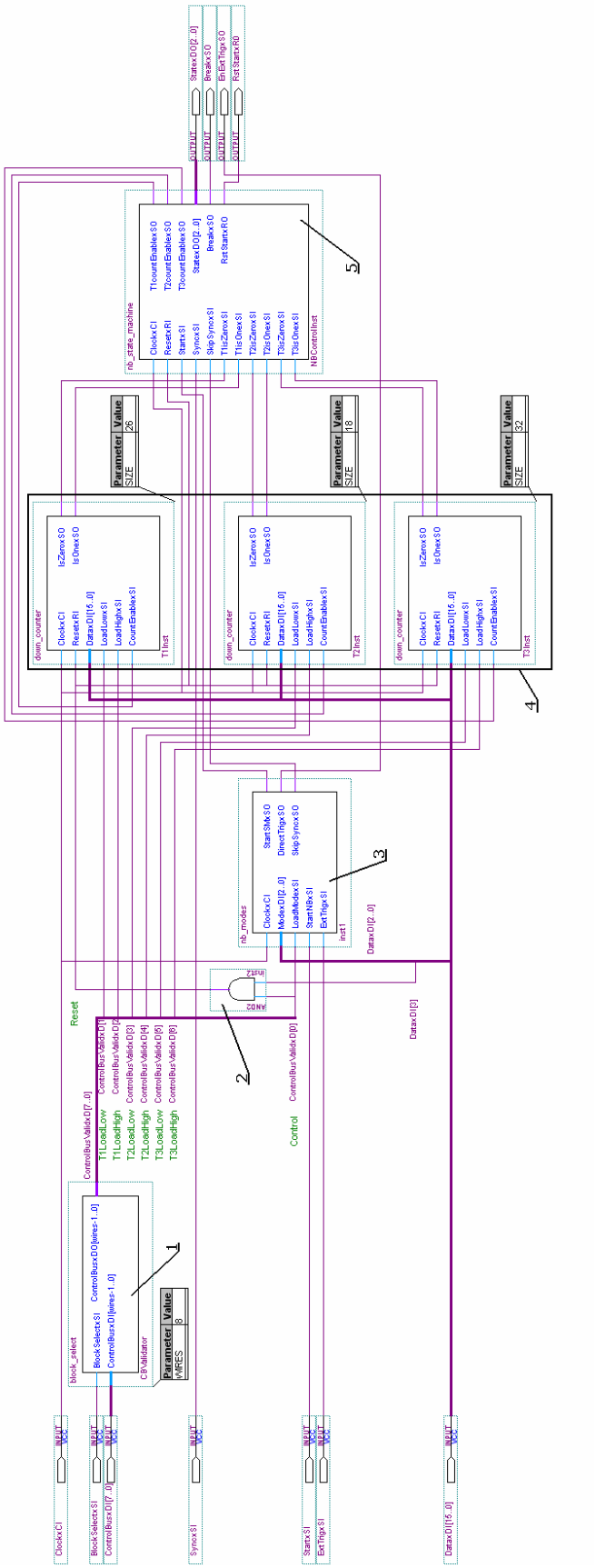

Figure 20 : NetBreaker diagram

8.1.6 NetBreaker structure

Besides the necessary **CBValidator** (1), this NetBreaker version contains:

- Logical AND (2) that allows a special reset function of the 4th data bit when a NetBreaker mode is set (see appendix B).
- **Modes** module (3) stores the selected NetBreaker mode and sets its outputs accordingly:
  - **StartSMxSO** either directly copies the start bit (the CSS and CS modes), or is a combination of the start bit and **ExtTrigxSI** (the ESS and ES modes) or is set to zero (the Ext. trig. mode).
  - **SkipSyncxSO** is active in modes that ignore sync (the CS and ES modes).
  - **DirectTrigxSO** is active only in the Ext. trig. mode.
- Next are the countdown counters (4), one for each of the measured times.
- At the heart of the NetBreaker is once again a FSM (5).

Two possible approaches to the time measurement have been tried here:

- Load the required times into registers, have up counters and compare their value against the registers.
- Load the required times directly into down counters and compare their value against zero (and one, because the FSM used here is Moore type and therefore requires the information about timer expiring one cycle before it actually happens).

The second approach requires half the DFFs of the first one and as this has been designed and tested on a Xiling 512 CPLD, DFFs proved to be the most critical resource.

You might have noticed that this NetBreaker doesn't generate a ready signal as mentioned in 8.1.4 but generates **StatexDO** instead. It is because the very latest requirements enforced significant changes in the NetBreaker design and while I feel this design should be in this thesis, I do not have the most up-to-date version prior to those changes.

In this version all the FSMs had hardcoded binary-encoded states and reported their current state instead of just a ready signal. I have included this obsolete design mainly because it had been the actual working solution for over six months. For the code of this obsolete FSM see appendix C.

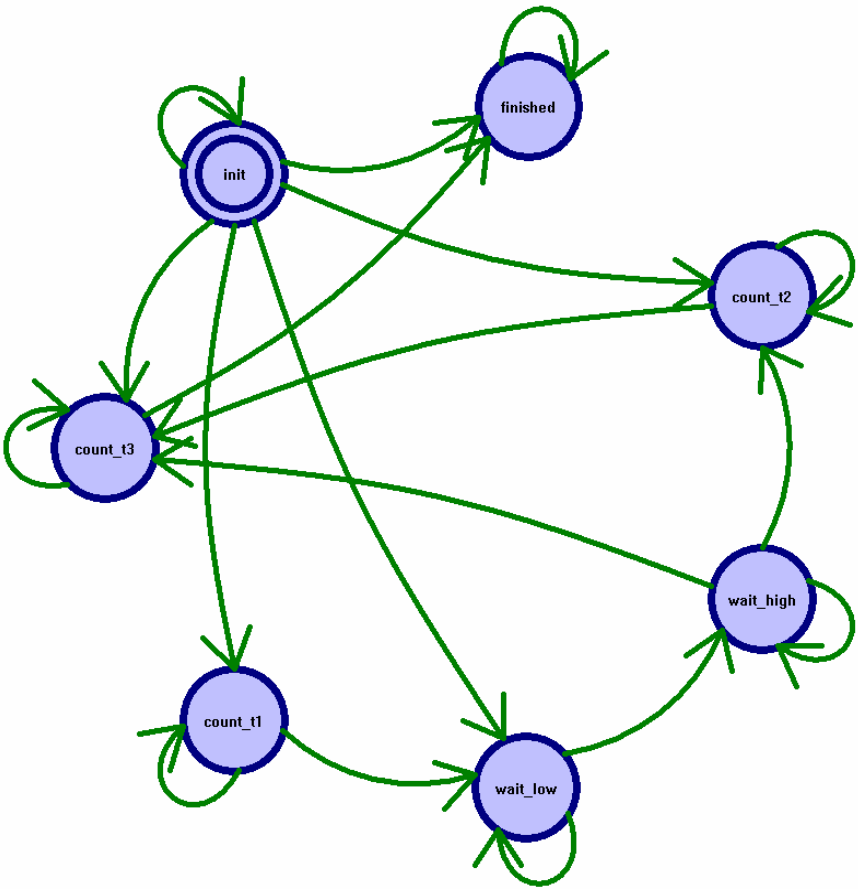

Figure 21 : NetBreaker finite state machine

8.1.6.1 NetBreaker finite state machine

This FSM can basically operate in two different modes. One is for NetBreaker modes with the full sequence (CSS and ESS): T1, Sync, T2 and T3. The other is for the Sync-less modes with the limited sequence: T2 and T3.

- **Init** state – Both modes start in this state and wait for the **StartxSI** signal. When the **StartxSI** signal comes there are several options:
  - T3 timer contains zero. No NetBreaking will happen anyway, so the FSM goes directly into the **Finished** state.
  - T3 is not zero and the NetBreaker isn't in a Sync-less mode. Then, based on T1 timer, it goes either into **Count_T1** state (T1 is not zero) or **Wait_Low** state (T1 is zero).
  - If the NetBreaker is in a Sync-less mode, two options present themselves, based on the state of T2 timer. If it is not zero, the FSM goes into the **Count_T2** state; otherwise it goes directly into the **Count_T3** state and starts NetBreaking.
- **Count_T1** state – the FSM stays in this state and generates **T1countEnablexSO** (the other Count states also generate their respective count enables) until the T1 timer comes down to one, the FSM then goes

into the **Wait_Low** state. The reason for comparing against one is, that with a count enable generated in this state only (i.e. Moore FSM) comparing against zero would mean staying in this state one cycle longer than is required.

- Example: T1 contains 1. The FSM goes into **Count_T1**, checks T1 against zero, and sets the next state to **Count_T1**. Next cycle. T1 contains 0, next state is set to **Wait_Low**, but the FSM has already spent two cycles in **Count_T1** instead of the required one cycle.
- Another possible solution would be tuning the FSM into a Mealy FSM, but at this point it was unnecessary.

- **Wait_Low** state – This state is very similar to its counterpart in the thermometer FSM. It ensures that T2 will be measured from **SyncxS**'s rising edge instead of a random point during its high state.
- **Wait_High** state – Again very similar to its counterpart, this time the FSM waits for the actual rising edge. When **Sync** goes into the high state, there are once again two possible transitions, based on the value of T2. If it contains zero, then the FSM goes directly into **Count_T3** and NetBreaking, otherwise it goes into **Count_T2**.
- **Count_T2** state – Measures the required time between **Sync's** rising edge and the start of NetBreaking.
- **Count_T3** state – This state generates the **BreakxSO** signal and measures the time for which it should be generated. When finished (again comparing against one) it goes into the **Finished** state.
- **Finished** state – The machine stays in this state until it is reset again.

One oddity that can be seen in this FSM (appendix C) is that the actual **BreakxSO** signal is set in the next state logic and, unlike all other output signals, gated. This is because the LAN Switches are connected to this signal only via combinational logic and any glitches unpredictably NetBreaking the net could prove fatal for the tests.

Another oddity is the way the other output signals are set, with all the signals for each state stored in a bit vector. This approach was supposed to raise the readability and the modifiability of the design, but proved to be more confusing than helpful and thus was abandoned in favor of a standard three process FSM.

8.2 Software

As stated in the analysis, changes to the PC software are minimal. Only two methods in the communication and a tab for NetBreakers in the GUI application are added.

8.2.1 Communication

With the Sync delay problem solved (7.2.1), the implementation of new methods has been quite straightforward:

SetNetBreaker takes the required mode and times, checks their validity, compensates T2 and then simply sends all these values into the device.

IsBreakerDone returns state of requested the **NetBreaker**, whether it is NetBreaking or not. In the first version this compared the returned FSM state to the known binary code for **Finished** state, the later versions directly evaluate the returned ready signal.

StartNB creates a 4 bit map of the required start bits and sends it into the device and returns.

StartNBBlocking does the same, but instead of returning immediately waits for all NetBreakers to finish which is checked by the **IsBreakerDone** method.

8.2.2 GUI application

Again, the implementation is based on the correct ID distribution amongst the dialog's controls. An automatic NetBreaker setup in a **for** cycle is very easy, as the difference between the RX and TX mode comboboxes is equal to both the difference between the TX and SWA mode comboboxes (and SWA with SWB) and the difference between the RX and TX T1 editboxes.

The only control where this posed a bit of a problem was the actual Modes combobox. Because their IDs are evenly spaced with other control's IDs in-between, we cannot use the **ON_CONTROL_RANGE** macro. So while there is just one method, **ModeChange**, that deals with the mode change (mainly disabling the T1 and T2 editboxes in the modes where they do not figure), there are four different methods registered as event handlers, one for each combo box, that call this method.

9. Detailed technical specification IV

The last modification of the requirements was again about the NetBreakers. The new requirement was that instead of just one NetBreak, there should be a sequence of several NetBreaks immediately following each other in a row (NetBreak repetitions). Then there should be an option to wait several Syncs (Sync repetitions), then again the same NetBreak sequence and so on, in a loop, for a specified number of repetitions (Total repetitions).

An example on the ***Figure 22*** shows NetBreaker in a CSS mode with sequence of three NetBreaks, followed by skipping two Syncs, the whole repeated twice.

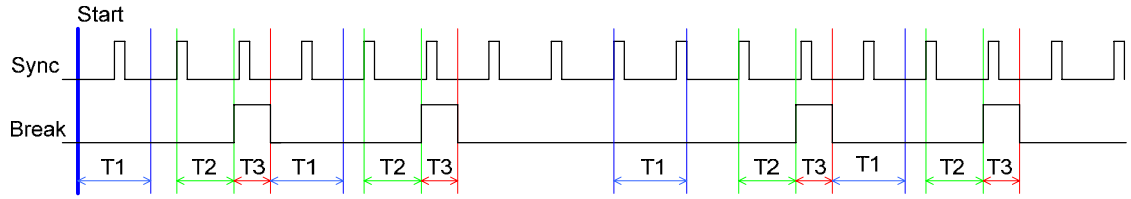

Figure 22 : NetBreaker – CSS mode with repetitions

Also, the USB board had been finished, so the final porting to the USB platform was required at this time.

10. Analysis IV

This analysis focuses on the problems encountered during the USB implementation and the possible approaches to the newest NetBreaker redefinition.

As the USB problems were mainly on the tempLab device side, their detailed description and solution is in my colleague's thesis [4].

10.1 USB analysis

We have encountered several problems here:

- The CDC drivers (3.2.2) were obviously unavailable for our AT43USB355 MPU.
- The documentation was less than ideal and we had only the USB Wizard [14] available as the source of USB support software.
- Most of the code generated by the USB Wizard is called, used in or calls functions from a library for which we had no source code available.
- Internet sources and forums we have searched yielded no useful code, mainly complaints at the USB Wizard generated code malfunctioning.
- There was no JTAG connector for code debugging, nor did we have any kind of a development kit to do the testing on.
- It was unclear what kind of transfers should we use, with the possible options being: Bulk and Interrupt as a type of transfer and HID and Mass storage as a USB class.

My colleague did some research of this topic and decided that the ideal combination would be a HID class device with Bulk transfer. Unfortunately, after several fruitless attempts it was discovered that the Windows HID class driver supports only Interrupt transfer.

We also asked Atmel for a sample code and received a very friendly reply with both Windows and AVR side sample codes.

10.2 NetBreaker modification

Tests with all times set to zero showed, that the minimal time for starting a NetBreaker, checking until it is done, loading and starting it again is 76ms. With Sync's intervals between 0.5 and 4 ms this proved to be an unacceptable delay.

Another option we considered was reloading the data from AVR, thus greatly reducing the reload time by the unpredictable communication delay between PC and the tempLab device. This option should be discussed in a greater detail in my colleague's thesis [4]. The main disadvantage was that the device would not be able to communicate for the whole NetBreaking sequence.

I came up with the last option after I studied the Max II structure, compared it to Xilinx FPGA structure and discovered that they are more similar than I would have expected for two different technologies (CPLD vs. FPGA).

CPLDs are known for having DFFs as their most critical resource, while FPGAs tend to have a lot of their Logic Elements (or equivalent, depending on the terminology of a given company) combinational only. Therefore have a lot of their DFFs available and unused.

After that, I have decided to take a closer look at the design's fitter report and, indeed, over half of the used Logic Elements was combinational only.

My idea therefore was to radically change the NetBreaker structure from three down counters to three registers with target times, three up counters for times and the same for Break, Sync and Total number of repetition counters.

The reason for this is that the down counters are destructive to the target value and therefore any restart of the counter requires reloading the original value. With combination of a register and a counter we can store the desired value in the register and then use it to either reload the counter (down counter) or to compare with the counter's value (up counter). I chose the later because the requirements should be same and it is more natural and easier to understand.

It could be argued that the counter for Total repetitions doesn't need the target value stored separately and therefore can be implemented as a loadable down counter. However, while this design resulted in less logic elements used, it also was much more complicated to route.

Currently no features can be added because then the design would be impossible to route anymore, so saving logic elements offers no advantage and the problematic routing results in approximately five times longer compilation time and slightly lower maximal frequency.

With the MPU and the CPLD project parts divided between me and my colleague, we decided to develop both versions simultaneously, using CPLD if it meets the requirements, MPU otherwise.

11. Solution IV

While the USB communication from the PC side is not trivial in its principle, it is very well defined, documented and with a lot of useful examples, so the actual implementation is very straightforward.

The new NetBreaker had been tried in two slightly different variations, namely the Total repetition counter implemented as a loadable down counter and an up counter with a register containing the target value (for the discussion see 10.2). I will describe only the final solution, containing an up counter with register. The other solution is clearly inferior for the tempLab device, because its routing takes several times more time with exactly the same functionality.

11.1 USB

I will describe the basic principle of USB communication initialization, an issue of plugging and unplugging USB devices and implementation of the desired communication functions into the existing communication DLL.

11.1.1 Initialization

Once it was determined we would be using HID class and Interrupt transfer type the implementation of the Windows part of the communication wasn't very problematic. This was mainly thanks to a very good and well documented example provided by Atmel (see appendix D), HClient example from Windows DDK [15] and an excellent article Making USB Device Drivers Easier by Stuart Allman [16].

The process of opening the desired device is in principle very simple. The system provides you with an array containing structures with information about all attached HID devices. You then inquire each of them for its **VID** and **PID** (Vendor and Product ID) until you find a device with the desired IDs. For more detailed description please refer to the aforementioned article [16]. Then you can use quite simply use **ReadFile** and **WriteFile** functions to read and write byte arrays to and from the device. The initialization process is always the same and I used the well documented one from the Atmel example.

11.1.2 Plugging and unplugging the USB

The examples also include methods that react to plugging and unplugging a device. Unfortunately, my application cannot react to Windows messages that report when there is a change in connected devices. Therefore, the communication has to be initialized by an explicit call of the **Connect** method and an unplugged device will not be recognized until the next attempt to communicate with the device.

The reason is that the while the GUI application can receive and process Windows messages, Python doesn't offer this option. I decided that a different behavior in such an important function would prove extremely confusing.

11.1.3 USB wrapper class

I decided to hide most of the initialization in a separate class. The class offers a **FindTheHID** method wrapping the whole initialization process. The other two public methods are **WriteReport** and **ReadReport** that send, respectively read, our protocol structure to, respectively from, the tempLab device.

It should be noted that the USB communication is master-slave and when **ReadReport** is called it waits until the device sends data or the operation times out.

Setting timeouts proved to be a problem, because the **SetCommTimeouts** function that should set timeout for the **ReadFile** operation obviously does not do so. The result is that despite any timeout set by **SetCommTimeouts**, the **ReadFile** method is always blocking and waits until the expected data arrive. This proved to be very problematic especially during debugging of the device side of the communication. When the device failed to respond with data, our PC application sometimes froze, in one case it froze the whole system and hard reset had to be used. The freezing seems to be absolutely random and probably tied to hardware, because while I had the problems in over half of the cases, my colleague had no trouble whatsoever.

As a timeout-less communication with possibility of crashing the system was definitely undesired, I searched for possible workarounds. I found that this issue is known, but as yet unaddressed by Microsoft. The solution is to call the **ReadFile** in a non-blocking mode using **OVERLAPPED** structure. The structure contains pointer to an Event that will signal state of the read operation. We can then use **WaitForSingleObject** method to wait for a specified time for this Event to become signaled. The result is that we have **ReadFile** with a timeout realized by two separate functions. Overhead of this solution is definitely higher than if the **ReadFile** method timeout worked correctly and if the issue is solved in the future, this wrapper should be updated.

Another problem we encountered but quickly solved is that the message received from USB is always one byte larger than the message sent. The reason is that the received message contains 1B header with identification of a receiving interface. We ignored the issue of multiple interfaces because the USB Wizard could generate only one interface. This manifested by the reserved receiving structure overflowing into a global variable in the MPU.

The only problem with discovering this error was the unavailability of any debugging besides sending variables onto a very limited number of LED that have been soldered to the board for this very purpose.

11.1.4 Using the USB wrapper

With USB ensuring correct arrival of the sent message in all but catastrophic scenarios we no longer needed our own timeouts, CRC checks, resending on failed message and other things implemented by the UART version of the **SendRecBlocking** method.

Usage of USB if extremely straightforward; calls **WriteReport** to send the command, **ReadReport** to receive the response, checks if any of these returned any errors and if so generates an appropriate error message. Finally it checks whether the response matches the command and returns result of this comparison.

The only other method that deals with the communication is the **Connect** method that now simply instantiates the USB wrapper and calls its **FindTheHID** method instead of calling the UART initialization methods.

The UART libraries have been removed from the project and are no longer used, as the MPU and thus the whole tempLab device does not support UART.

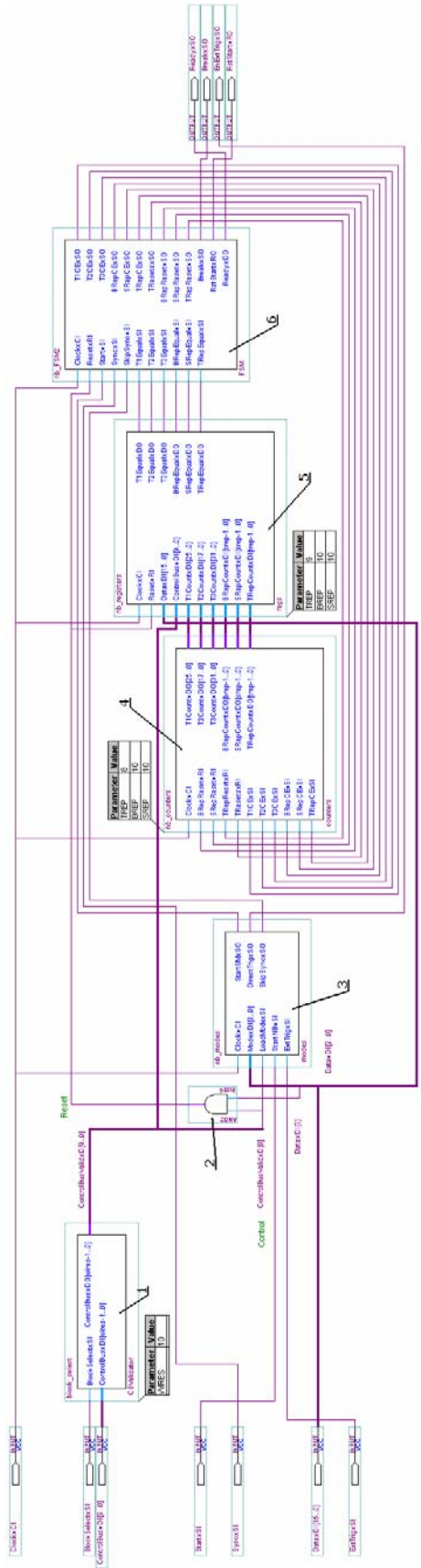

Figure 23 : NetBreaker rev 2. diagram

11.2 NetBreaker

The desired goal is to have all three repetition counters with a 16bit range. This would allow sequence of 0-65,535 NetBreaks, followed by skipping 0-65,535 Syncs and this repeated up to 65,535 times. Unfortunately this does not fit into the used CPLD, so the current version supports only up to 1023 Breaks and Syncs and this as a whole repeated up to 255 times.

I have tried using both a loadable counter and an up counter with register for Total repetitions counter. The limiting factor here seems to be the routability of the design, not the logic elements. The loadable down counter proved to lead to a less routable design with no advantage from the gained logic elements, so I have opted for the up counter and register solution.

The components 1-3 of this block are explained in the original NetBreaker description (8.1.6).

- **Counters** (4) component contains all six up counters for both the timers and the repetition counting. All the counters have separate count enables. T1-3 timers have a common reset signal; the other counters have each their own.
- **Regs** (5) contains registers with the target values for all counters and a set of comparators to compare them with counter states.
- Description of the Finite State Machine (6) is in a separate chapter.

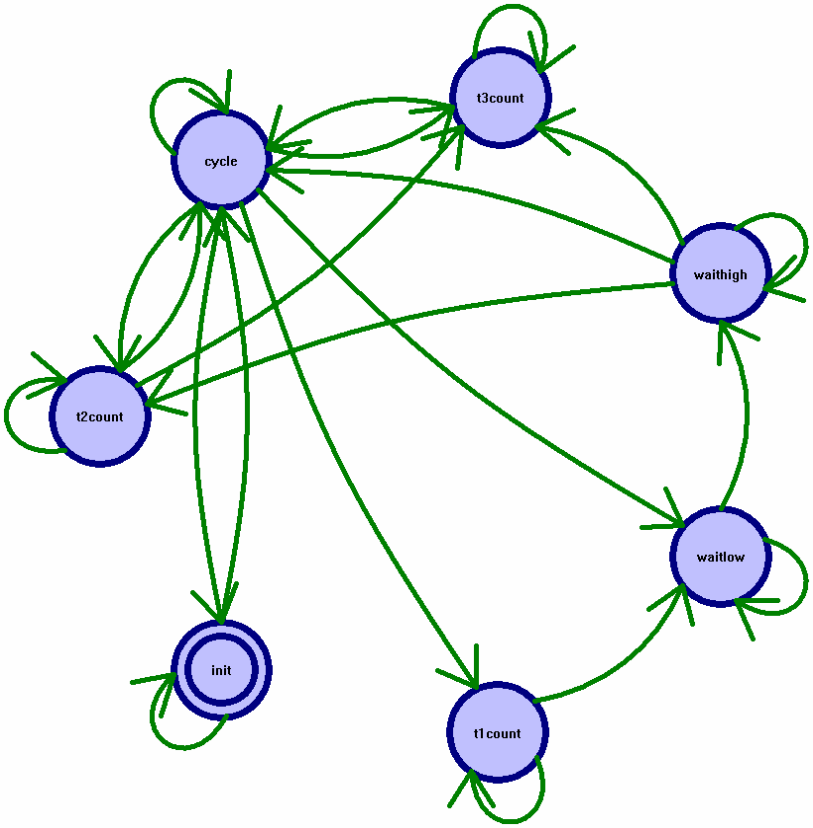

Figure 24 : NetBreaker finite state machine (cycles)

11.2.1.1 NetBreaker finite state machine

This FSM contains a **BreakSync** signal that determines whether the current sequence is NetBreaking or Sync-skipping. When **BreakSync** is in log 1, the FSM performs the standard NetBreaking sequence as described in 8.1.6.1. When it is in log 0, the FSM goes only through the **WaitLow** and **WaitHigh** states, therefore simply counting Syncs.

When either of these FSM sequences finishes, the appropriate counter (Break and Sync repetitions respectively) is incremented. When the counter reaches the target value (expires), it is reset and the **BreakSync** signal is inverted. When Sync expires, the **BreakSync** goes from log 0 to log 1, the Total repetitions counter is incremented and when this counter expires, the whole NetBreaker sequence ends.

There are two exceptions to this general rule; both occur when one of the target values equal to zero. When either Sync or Break target repetitions are zero, the **BreakSync** signal keeps the same value all the time, never switching to the sequence that should be performed zero times.

When Sync repetitions are set to zero, the Total counter is incremented when the Break counter instead of the Sync counter expires.

The main change besides the repetition mechanism is that the whole FSM had been changed from Moore into Mealy type. This is because the counters are now counting upwards instead of down to zero. It would be extremely impractical to compare the value against target value minus one. And if this subtraction was done in the PC application, it would be impossible to tell when target value is zero.

Finally, as the machine required an almost complete redesign because of the repetition sequences, redesigning the NetBreaking cycle from Moore to Mealy meant almost no extra time.

The states and their transitions:

- **Init** state – This state is identical to the previous version, therefore it just waits for a start signal.
- **Cycle** state – This is the most complicated state, because it decides what will the FSM do next. There are several possibilities:
  - The Total repetitions counter expired, the whole NetBreaking loop is completed and the machine goes back into the **Init** state.
  - The FSM is NetBreaking (**BreakSync** is log 1) and the Break repetitions counter expired. The **BreakSync** signal is inverted (to log 0), the Break counter is reset and the FSM goes again into **Cycle** state.
  - If the previous is true, but the Sync repetitions counter is also expired (i.e., the target number of Sync repetitions is zero), then the **BreakSync** signal stays in log 1 and the Total repetitions counter is incremented. The next state of the FSM is **Cycle**.

- The FSM is Sync-skipping and the Sync repetitions counter expired. The **BreakSync** signal is inverted (to log 1), the Sync counter is reset, the Total counter is incremented and the FSM goes again into the **Cycle** state.
- If the previous is true, but the Break repetitions counter is also expired (i.e. the target number of Break repetitions is zero), then the **BreakSync** signal stays in log 0 and Total counter is incremented. The next state of the FSM is **Cycle**.
- If neither of these is true, then if the FSM is NetBreaking it starts the standard NetBreaking sequence as described in 8.1.6.1 by going into the **T1Count** state. If the FSM is Sync-skipping it goes into the **WaitLow** state, waiting for the next Sync.

- **T1Count** state – FSM stays in this state and generates a **T1CExSO** signal until the T1 timer is equal to the target value, the FSM then goes into the **Wait_Low** state.
- **Wait_Low** state – This state waits for the **SyncxSI** to go into low state and therefore ensures that the FSM will react to the rising edge of Sync, not just its high state.
- **Wait_High** state – Waits for the **SyncxSI** to go into high state, thus marking the rising edge. There are two possible results when the Sync rising edge is recognized, depending on the **BreakSync** signal:
  - When the FSM is Sync-skipping, then the Sync repetitions counter is incremented and the FSM goes into the **Cycle** state.
  - When it is NetBreaking, the FSM goes into either the **T2Count** state, or, when the T2 counter is already expired, directly into the **T3Count** state and starts NetBreaking (i.e., generates the **BreakxSO** signal). In case, if T3 is also expired then all the Time counters are reset and FSM goes into the **Cycle** state.
- **T2Count** state – Measures the required time between **Sync**'s rising edge and the start of NetBreaking. When the timer expires the FSM goes into the **T3Count** state, if the T3 is also expired it goes into the **Cycle** state instead.
- **T3Count** state – This state generates the **BreakxSO** signal and measures the time it should be generated for. When it is finished, the FSM goes into the **Cycle** state.

12. Testing

Both software and hardware had their functionality tested in separate tests, testing each component of the solution individually and in long run tests, testing the solution as a whole.

12.1 Software tests

The main tested feature was the communication between different PC modules, where the tests focused on correct passing of parameters between the modules.

All the COM methods had been called from both GUI and Python to test whether the parameters are passed correctly. This was tested for typical, maximal, minimal and out-of-range (both larger than maximal and lower than minimal) values. We have never encountered any error, but we still decided to implement a Ping function that will check whether the device is working independently on any passed values.

Besides this, the software was tested only in the long run tests described in a separate chapter.

12.2 CPLD tests

All the CPLD blocks had been simulated both separately and as a part of the complete design and then tested in the long run tests.

12.2.1 Basic Input Unit

The **BIU** had been simulated in full range of required addresses and it generates correct **BlockSelectxDO** and **ControlBusxDO** signals in the whole range.

The main results of the timing analysis simulation were the minimal required times for the input signals to be active, to ensure the BIU recognizes the communication and reacts correctly. The result is that when a MPU is writing into the CPLD, the data from the bus are read no later than 83.33 ns after both CS and WR signals become active. When MPU reads from the CPLD, the data on the bus are valid no later than 104.17 ns after both CS and RD signals become active.

It needs to be considered that these values are based on timing analysis and take into account neither the delays between the CPLD and the MPU, nor the delays in the input and output buffers of the MPU. Therefore, the actual control signals should be sufficiently longer.

12.2.2 Relays block

The relays block is represented by a simple register with count enable, so no problems had been expected. It was simulated for several values and then tested in the long run tests.

12.2.3 Thermometers block

The thermometers block had to be simulated with probes running on higher frequencies than the real 1-4 kHz. The reason was that the simulations using the real probe frequencies took unacceptably long time on the computers we had available.

We have therefore decided to use a PWM generator and compare the results from thermometers block measurement with the Duty Cycle set on the generator.

Both simulations and the tests proved the thermometers block should be able to measure the PWM signal from thermometers probes in full range of frequencies and temperatures.

12.2.4 NetBreaker block

The NetBreakers could be simulated only for very low values, because simulating the full range of values would take extremely long on the computers we had available. I have therefore focused on simulating low values of times and repetitions. The correct response when the required times are equal to zero is almost impossible to tell outside the simulation, because any signal measured by logic analyzers is also influenced by delays of input and output buffers.

We have used a logic analyzer to test the functionality for the typical and large values. The logic analyzer had also been used to determine the minimal delay between a rising edge of Sync and NetBreaking (with T2 = 0) as discussed in 7.2.1.

12.3 Long run tests

The long run test had been done on several of the latest revisions. It always had an almost identical structure, testing all the functions simultaneously:

- Relays (setting) – Relays had been set to a random value; their state had then been inverted, followed by another value and so on. This had been done every thirty seconds for the first two hours of the test, then every five seconds for twenty minutes and then every minute for the rest of the test.
- Relays (reading) – Relay state had been continually read and the value compared to the last set value.
- Thermometers – All thermometers had been continually measured and the value stored. We didn't use any calibration or error correction on the measured values. The 24 hour version of the test did the continual measurement only for the first two and half hour then the measurement was done only every minute.
- NetBreakers – NetBreakers are almost impossible to test in this way, so we just loaded a range of typical values into them, kept restarting them whenever a NetBreaker finished and then only checked if the NetBreaker

was run for an expected number of repetitions. That is, run sixty times when set to a one minute cycle and the test ran for one hour

The long run tests revealed two major problems.

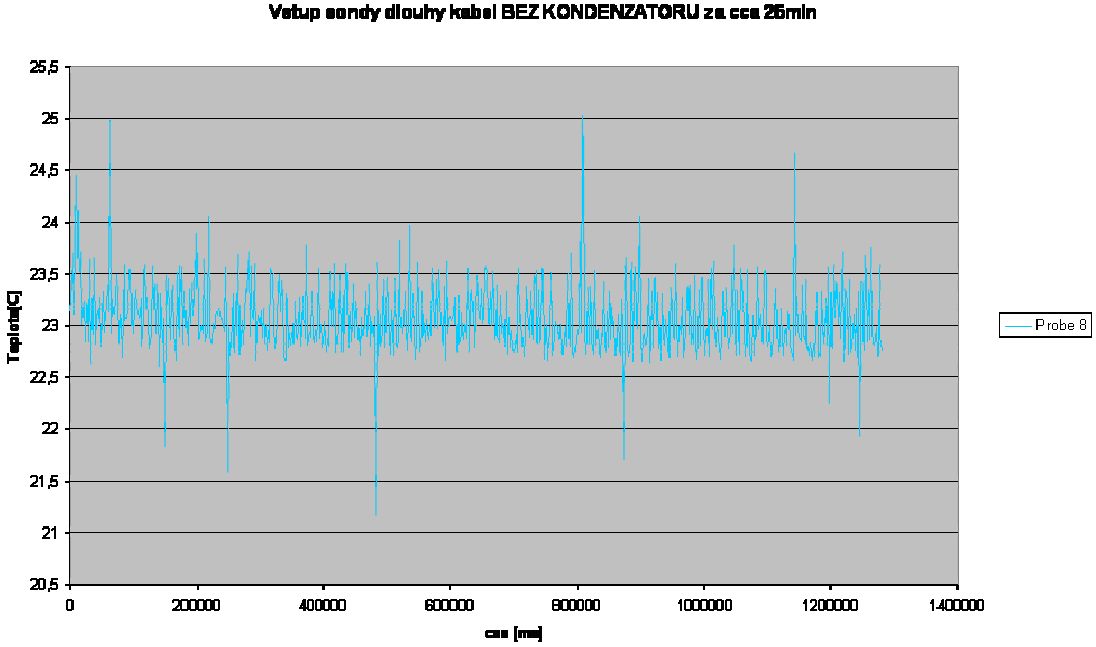

Figure 25 : Thermometer on 6m cable with no capacitor

The first problem was with the thermometer probes that kept returning unexpectedly unstable values well off from the claimed ±0.7°C precision. Results of a short 25 minute test used to precisely analyse the problem are on ***Figure 25***. Sometimes they even returned obviously incorrect values, -30°C when the average temperature was 26°C. This was apparently caused by unstable power source, for the details please refer to my colleague's thesis [4]. The problem was solved by adding a capacitor to stabilize the thermal probes' power; the measurement then became much more stable, with the difference between two following values within ±0.1°C (***Figure 26***).

The second problem was that during a 24 hour test, the device stopped responding for almost 70 seconds and then started working again. Unfortunately, we couldn't reproduce this error on demand to perform any measurements. Our best guess is that the problem lied in the communication between the MPU and the CPLD. The control signal were active for a period very close to the minimal critical time and the clocks of the MPU and the CPLD probably got into such a phase shift, that the period was insufficient and the CPLD didn't recognize the signals were active and not just glitches. This was on the ATMega 128 development board; the final version is now using longer control signals. For more details please refer to my colleague's thesis [4].

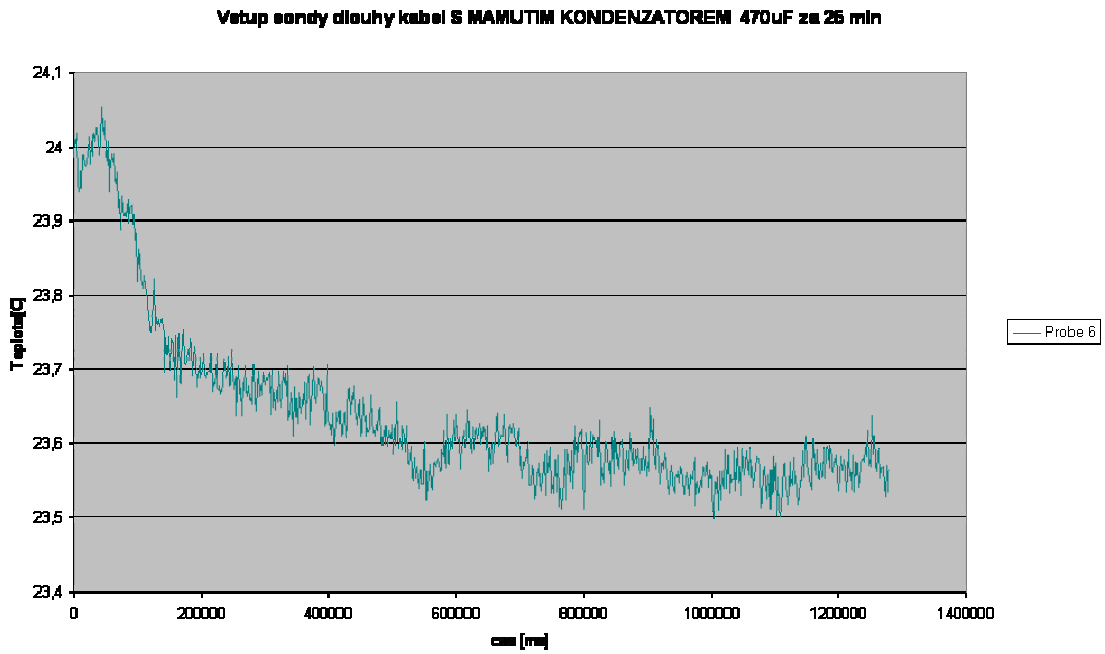

Figure 26 : Thermometer on 6m cable with a 470µF capacitor

13. Conclusion

The tempLab device with its PC software fulfills all the set requirements, both the original ones and the requirements that appeared during the development.

The new requirements include:

- Independent NetBreaking on all four NetBreakers.
- Two NetBreakers used for switching an input between two outputs instead of just connecting and disconnecting it to a single output.
- NetBreaking as a sequence, not just a single NetBreak. The device currently allows a sequence of up to 1023 NetBreakes, followed by waiting for up to 1023 Syncs, this as a whole looped up to 255 times.

In addition to this, the device offers a much higher precision of NetBreaking. Originally, the times were to be specified in microseconds, the tempLab device now allows specifying the times in 21ns steps. After figuring in the uncertainty of the reaction time of the used LAN switch, the precision should be about 30ns, therefore still far bellow the original requirement. This does not influence the requested time ranges in any negative way, T1 can be up to 1.39s, T2 up to 5.46ms (requested 4ms) and T3 up to 89.4s (requested 60s).

All the required functionality is implemented in a CPLD that provides not only the aforementioned NetBreaking, but also controls relays that allow turning up to sixteen tested devices on and off and measures duty cycle of a PWM signal generated by any of the up to sixteen thermometers that can be connected to the tempLab device and used to measure temperature in the tested devices.

Another important part of the tempLab device is a MPU with an integrated USB controller that passes commands and responses between the PC software and the CPLD.

On the PC side there is a COM DLL that communicates with the tempLab device and offers all the methods necessary for using the tempLab device to any application capable of calling COM DLL methods. The project includes two such applications, first is a GUI application that can be used for an easy and user-friendly controlling and testing of the tempLab device, but is not very suitable for actually running PROFINET tests. For the PROFINET tests, there is a Python object that can be easily used by any Python script used for the tests.

The tempLab device has undergone series of tests and after few minor modifications it was decided that the device fulfils all the requirements. The tests focused mainly on the stability of the device as a whole, but also tested all the required precisions and timings.

During the development I have discovered two errors in the used development tools. The first one was a malfunctioning *midl.exe* in Visual C++ SP6 that had to be replaced by *midl.exe* and *midlc.exe* from Platform SDK. The second was that using

SetCommTimeouts to set timeout of a ReadFile method when reading from a HID class USB didn't have any effect and workaround had to be used to actually implement a timeout.

The development took almost fourteen months, not only because the project proved to be larger and more complicated than was originally expected, but also because the NetBreaker specification had been changed several times and as a result we had to abandon an almost finished software-based solution and start anew on a hardware-based solution using a CPLD.

References

[1] PROFINET – Introduction. URL: http://ia.edisonautomation.com/networking/ProfiNet.html

[2] Charon 2 – Ethernut embedded Ethernet module. URL: http://www.hwgroup.cz/products/charon2/index_en.html

[3] Ethernut. URL: http://www.ethernut.de/en/

[4] Bernatík, Vít: *TempLab – tester sítě PROFINET* (bachelor thesis). ČVUT, Praha 2006

[5] WizNET: W3100A – a hardwired TCP/IP chip. URL: http://www.iinchip.com/wiznet/product_assp.html

[6] Future Technology Devices International Ltd. URL: http://www.ftdichip.com/FTProducts.htm

[7] Atmel Corporation: AT43USB355 (product informations). URL: http://atmel.com/dyn/products/product_card.asp?part_id=2573

[8] Atmel Corporation: *Migrating from RS-232 to USB Bridge Specification*, Rev. 4322A-USB-01/04. URL: http://atmel.com/dyn/resources/prod_documents/doc4322.pdf

[9] Atmel Corporation: AT76C713 (product informations). URL: http://atmel.com/dyn/products/product_card.asp?part_id=3556

[10] Don Box: *Essential COM*. First edition, USA, Addison Wesley Longman, Inc., 1997. ISBN 0-201-63446-5

[11] Brändli, Matthias: *Naming Conventions*. ETH Zurich, Switzerland. Last change: June 14th 2005. URL: http://dz.ee.ethz.ch/support/ic/vhdl/vhdlnaming.en.html

[12] Bečvář, Miloš: *VHDL description of digital circuit – rules for writing a synthesizable code*. ASICentrum, Czech Republic, July 8th 2003 (last modified November 18th 2003). URL: http://service.felk.cvut.cz/courses/X36PNO/labs/VHDL_synth_rules.pdf

[13] Altera: *Max II Device Handbook*. URL: http://www.altera.com/literature/hb/max2/max2_mii5v1.pdf

[14] AT43USB35x Development Kit. URL: http://atmel.com/products/USB/forms/docs/AT43DK355_Installation_Package_JUN282005.zip

[15] Microsoft Windows Server 2003 SP 1 Driver Development Kit

[16] Allman, Stuart: *Making USB Device Drivers Easier*. URL: Not available anymore. For a downloaded PDF version see appendix E.

[17] WizNET: EVBAVR ATMega 128 Evaluation Board (product informations). URL: http://www.iinchip.com/wiznet/product_evbavr.html

[18] Smartec Sensors: SMT160-30 (datasheet). URL: http://www.smartec.nl/pdf/DSSMT16030.PDF

Appendixes

A Communication protocol signals – \Appendixes\A\CPLD_Bus_Cycle_3.vsd

B Communication protocol commands – \Appendixes\B\commands.xls

C Obsolete (pre-cycles) NetBreaker FSM – \Appendixes\C\nb_state_machine.vhd

D The PC HID example by Atmel - \Appendixes\D\

E Making USB Easier article - \Appendixes\E\Making USB Device Drivers Easy.pdf

CD Content

\	
├──Appendixes	Electronic appendixes
├──doc	Documentation for C++ projects
├──install	Installation files
├──src	
│ ├──CPLD	CPLD source files (Quartus 5.1 SP2)
│ ├──Python	Python source files (Python 2.3.4)
│ └──VisualC	Visual C++ project files (MS Visual C++ 6.0 SP6)
└──thesis	PDF and MS Word versions of the thesis text

List of Figures