GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN

Fakultät für Physik

Bachelor's Thesis

Entwicklung einer Softwareanwendung zur I2C Konfiguration des CERN lpGBT Chips für die Ausleseelektronik des ATLAS LHC Detektors

Development of a software application for I2C configuring of the CERN lpGBT chips for the ATLAS LHC detector readout electronics

prepared by

Matthias Drescher

from Gütersloh

at the II. Physikalischen Institut

Thesis number: II.Physik-UniGö-BSc-2020/01

Thesis period: 30th March 2020 until 6th July 2020

First referee: Prof. Dr. Arnulf Quadt

Second referee: Priv.Doz. Dr. Jörn Große-Knetter

Abstract

Diese Bachelorarbeit beschreibt eine Java GUI Anwendung zur Konfiguration des lpGBT Transceivers, der bei der Phase-II Aufrüstung des ATLAS Spurdetektors eingesetzt wird. Zur Umsetzung wird das CERN USB-I2C Dongle als zusätzliche Hardware benötigt. Anstelle des lpGBT ASICs, welches im späteren Anwendungsfall benutzt wird, wird das Programm mithilfe des lpGBT FPGA Emulators getestet, der in dem Xilinx Ultrascale+ KCU116 Evaluation Kit implementiert ist. Die Tests bestätigen, dass die Anwendung eine oder mehrere lpGBT Instanzen über denselben Bus konfigurieren kann, erlauben jedoch keinen Aufschluss über die Fähigkeit der Anwendung, die sonstigen Module zu steuern oder die Sicherungen des lpGBT ASICs zu brennen.

Stichwörter: lpGBT ASIC, FPGA Emulator, Konfigurationssoftware, ITk Pixel Auslese

Abstract

This bachelor thesis describes a Java GUI application for the configuration of the lpGBT transceiver, which is part of the ATLAS Phase-II inner-tracker upgrade. For this purpose, a CERN USB-I2C dongle is required as additional hardware. Tests are carried out with the lpGBT FPGA emulator hosted on a Xilinx Ultrascale+ KCU116 development board, instead of the lpGBT ASIC used in the real setup. The tests successfully validate the programmer GUIs ability to configure one single or multiple lpGBT instances over the same bus, but cannot test its ability to control the additional modules and fuses included in the ASIC.

Keywords: lpGBT ASIC, FPGA emulator, configuration software, ITk pixel readout

Contents

1 Introduction

To advance particle physics, collider experiments with high center of mass energies producing large amounts of data to improve statistics are needed. The ATLAS experiment at the Large Hadron Collider (LHC) is one of these experiments. Alongside CMS, it tests the Standard Model of particle physics and searches for physics beyond it. With the planned upgrade of the LHC to the High Luminosity Large Hadron Collider (HL-LHC), increasing its luminosity to $\mathcal{L} = 5 \times 10^{34}\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$ and its center of mass energy to $\sqrt{s} = 14\,\mathrm{TeV}$ [1], the ATLAS experiment needs to upgrade as well to benefit from it. The HL-LHC will be installed during the long shutdown 3 (LS3) from 2025 to mid-2027. One of the changes in the Phase-II ATLAS upgrade are the newly developed inner-tracker (ITk) pixel and strip detectors. The radius of the innermost layer of the pixel detector is reduced from an average of 50.5 mm to a maximum of 39 mm, to reduce the lever arm of the measured trajectories. Also, their resolution is increased, to be able to resolve the different collisions within a bunch crossing from one another. For example, the ITk pixel sensors decrease the pixel size to $50\,\mu\mathrm{m} \times 50\,\mu\mathrm{m}$ compared to the current value of $50\,\mu\mathrm{m} \times 400\,\mu\mathrm{m}$ for most of the sensors. Of course, with finer resolution the amount of sensor data to be transmitted outside the detector and processed also increases. That is why also a new transceiver chip, the Low Power Giga Bit Transceiver (lpGBT), is being developed for use with the ITk pixel detector, amongst others. Increased radiation levels due to the higher number of collisions are a key design aspect for the new hardware.

The goal of this thesis is to describe the development and testing of a graphical user interface (GUI) application for configuring the lpGBT over the I2C protocol, called "lpGBT programmer software" [11]. For this, first the backgrounds necessary for understanding the different parts of the project, as well as more details about the context in which the lpGBT will operate, are presented in Sec. 2. Afterwards, the hardware equipment used in the development process and the hardware setup used for the real world configuration application will be discussed in Sec. 3. The programmer software itself is detailed and discussed in Sec. 4. It covers the programmer software specifications, its structure and also a view on the programmer interface and how it is used to configure the lpGBT. Finally, the different test setups and procedures are explained in Sec. 5, and their results are shown, before a conclusion is given in Sec. 6.

2 Background

2.1 Large Hadron Collider

The Large Hadron Collider (LHC) [2] is a proton-proton collider located inside a 26.7 km long underground tunnel near Geneva. It was built as the successor of the Large Electron-Positron (LEP) collider, because the synchrotron radiation losses are significantly lower for the heavier protons as compared to those of the electrons and positrons ($E_{\text{loss}} \propto m^{-4}$), and protons are easy to produce in large numbers compared to the other heavy, charged and stable particles or antiparticles. The resulting centre of mass energy of currently 13 TeV, which is the highest to date, allows not only for testing the standard model of particle physics, but also for discovering new particles beyond the standard model, which are expected to have higher masses and therefore need a high centre of mass energy to be produced. The two separate beam pipes, in which the protons are accelerated in opposite directions, meet at four interaction points, at each of which a collider experiment is hosted. Currently, the four main experiments at the LHC are the ATLAS, CMS, ALICE and LHCb experiments. Because of the environment, in which the programmer software was developed, and not because of any restrictions in using the lpGBT or the programmer software for different experiments, the ATLAS experiment is the context, in which the lpGBT is discussed.

2.2 The ATLAS experiment

2.2.1 General

The ATLAS detector [3] is a general purpose detector. It measures the particles created from the proton-proton collisions by their interactions with the detector material in different cylindrical layers around the beam pipe. By combining the information of these different layers, its type, energy and momentum may be reconstructed.

The innermost layer is the inner tracking detector. Since, for small distances to the beam pipe, measurement errors of the particles position have a larger impact on the

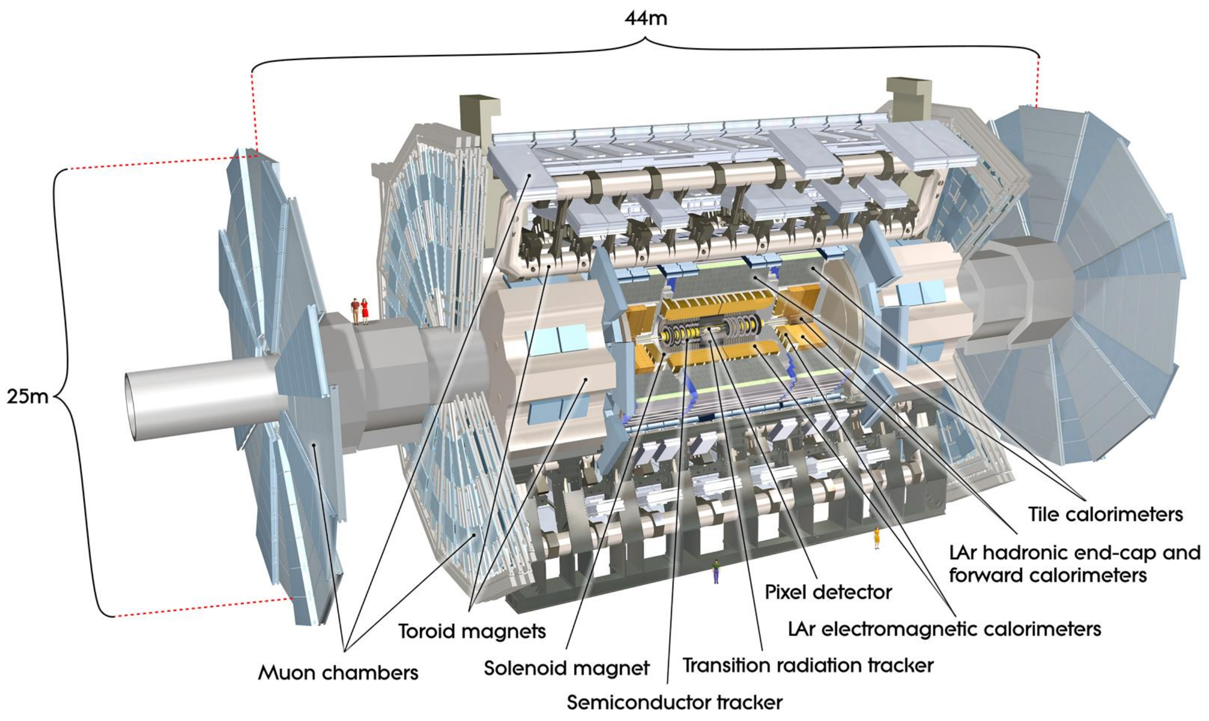

44m
25m
Tile calorimeters
LAr hadronic end-cap and forward calorimeters
Pixel detector
LAr electromagnetic calorimeters
Toroid magnets
Muon chambers
Solenoid magnet
Transition radiation tracker
Semiconductor tracker

Figure 2.1: A computer generated cross-sectional view of the ATLAS detector. © CERN

measured direction of the particle because of the small lever arm, it consists of three layers with increasing resolution towards the beam pipe. Going from the beam pipe outwards, the particle first encounters the pixel detectors, then the strip detectors, and finally a transition radiation detector. Pixel and strip refer to the shape of the cells on the semiconductor sensors. Solenoid magnets around the inner detector create a 2 T magnetic field parallel to the beam axis. With this, a charged particles trajectory is not a line, but rather a helix, and the particles transverse momentum vector can be calculated from the curvature of the helix. Neutral particles are not detected, as they do not leave ionisation tracks in the semiconductor sensor.

After the inner tracker follow, in order, the electromagnetic and hadronic calorimeters [4]. In these, the respective particles react with the detector medium, creating more particles doing the same. This leads to a particle shower, where the energy of the original particle is distributed over more and more particles. The summed sensor signals along the showers path is then a measure for the original particles energy. The electromagnetic calorimeter detects photons, electrons and positrons, which generate the showers by a combination of $\gamma \rightarrow e^+e^-$ pair creation and Bremsstrahlung. The hadronic calorimeter detects hadrons, for which the main processes are inelastic collisions with the nuclei of the detector material.

The calorimeters in the ATLAS detector are of sampling type, meaning that the passive material generating the shower and the active material, in which the measurement is performed, for example by collecting the ionisation charge, are not the same. This is the more compact and cheaper solution, but has the drawback, that only parts of the shower can be probed. Because different detector regions require different calorimeter designs, the calorimeters are divided into a barrel part around the beam axis, an end-cap part closing the barrel and a forward calorimeter (FCal) for the even higher rapidity regions. These different regions can be see in the cross-sectional view of Fig. 2.1. The hadronic version of the barrel calorimeter is the tile calorimeter. The location inside the detector influences the shape of the calorimeters and the materials used, which are listed in Tab. 2.1. For example, most of the calorimeters use liquid Argon as the active material, which can be easily exchanged, making it a good candidate for the high radiation zones. But the hadronic tile calorimeter can use the cheaper scintillating tiles, as it is further away from the beam. Similar to the tracking detector, the calorimeters increase in granularity towards the beam axis.

Type		Barrel	End-cap	Forward
Electromagnetic	Active material	Liquid Argon	Liquid Argon	Liquid Argon
	Passive material	Lead	Lead	Copper
	Rapidity region	$\lvert\eta\rvert < 1.475$	$1.375 < \lvert\eta\rvert < 3.2$	$3.1 < \lvert\eta\rvert < 4.9$
Hadronic	Active material	Scintillator	Liquid Argon	Liquid Argon
	Passive material	Steel	Copper	Tungsten
	Rapidity region	$\lvert\eta\rvert < 1.7$	$1.5 < \lvert\eta\rvert < 3.2$	$3.1 < \lvert\eta\rvert < 4.9$

Table 2.1: The different parts of the ATLAS calorimeter, and the materials they use. The scintillator is Polystyrene in combination with a wavelength shifter.

The only particles that remain after the calorimeters are muons, as they are not hadronic, but also have a mass that is too high for showering in the electromagnetic calorimeters, because the intensity for Bremsstrahlung is $\propto m^{-2}$ (calculated classically). They do, however, leave ionisation tracks. This is used in the muon chambers, the outer-most layer, which measure the muon tracks and momentum inside a magnetic field similar to the tracking detector. Toroidal magnets generate a magnet field in the muon chambers, with field flux densities of 1 T in the central region and 0.5 T in the end-caps. The magnetic field lines follow the azimuthal angle ϕ around the beam axis, as opposed to the field lines of the inner solenoid, that generates a magnetic field with field lines parallel to the beam axis.

2.2.2 Pixel readout electronics

A closer look shall be given at the readout system of the upgraded inner-tracker (ITk) pixel detector [5]. The inner-tracker upgrade including the readout system will be installed alongside the HL-LHC in LS3. Regarding the readout system, first laboratory tests with a full readout chain are prepared just at the time of writing. The absence of such setup in the testing phase of the programmer software directly influences the testing setup described in Sec. 5.

The readout chain starts at the pixelated semiconductor sensors, where the ionisation tracks of the to be detected particles are created. The separated ionisation charges generate a current in the pixel electrodes. For reading out the current signal of the passive sensors, a separate, active front-end readout chip is necessary. It contains readout pixels mirroring the sensor pixel layout, so that each pixel on the sensor is connected to a readout pixel, when the two chips are connected. The connection between the pixels is established using bump-bonding technology. The front-end chip also contains common logic for reading out the pixels.

One, two or four front-end chips using one sensor chip of corresponding size are grouped into single, dual and quad modules, respectively. Multiple front-end chips are then connected to an aggregator chip, which merges the front-end data lines into one single data line. This data line finally connects to the off-detector counting room for data acquisition via optical links.

At the ATLAS experiment, the proton bunches collide with a frequency of 40 MHz. Reading out every single event would yield data rates in the order of many PB/s, and it would be impossible to store and analyse the data. To reduce this frequency without losing the rarely occurring events one wants to look at, a trigger system selects which events to store. This is done in multiple stages, where the first stage, the Level-1 (L1) hardware trigger, only uses a crude but very fast analysis, giving the second stage, the High-Level Trigger (HLT), more time to take a decision. The L1 trigger may still need up to 2.5 µs to decide, so the event data has to be buffered on the read out chips until the L1 trigger decision is known. This further complicates the readout chain, as it is now necessary to have bidirectional communication between the front-end chips and the counting room for sending the trigger signals, instead of simply sending the sensor data one way. Apart from the trigger signals, also the front-end chip configuration uses the downlink data path: Because the front-end chip, due to space restrictions, cannot protect the pixel configurations from single event upsets (SEUs) caused by the radiation, the pixel configuration has to be periodically refreshed from the outside. This technique is called "trickle configuration".

For the ATLAS upgrade, the RD53 [6] is used as the front-end chip, and the lpGBT, which is described in Sec. 2.6 and the target of the configuration software, is the data link.

2.3 ASICs and FPGAs

Certain applications require special integrated circuits with unusual specifications, that are not available commercially. These then have to be designed manually by the user, but as a result can be custom tailored to the use case. The production of the chips according to the design is then done by a hardware manufacturer. The result is an application specific integrated circuit (ASIC). Inside the ASIC, transistors are directly connected to form logical gates and higher level structures, which then make up the ASICs functionality.

An alternative to ASICs are field programmable gate arrays (FPGAs), which are configurable integrated circuits: As opposed to the ASIC, the chips functionality is not implemented in the FPGAs transistors directly. Instead, they form configurable logic blocks with configurable connections between them. A simple example of a logic block is a look up table (LUT), that contains the desired output logic level (high, low) for each combination of input logic levels. By setting the values of the table, a LUT may emulate any logic gate. Thus, together with the configurable connections between the logic block inputs and outputs, the same higher level structures as present in the ASIC may be created.

Depending on the memory technology used for storing the connection command values inside and amongst the logic structures, the FPGA can be one-time programmable (OTP) in the case of anti-fuse connections, or reprogrammable in the case of using RAM-based or EEPROM (flash) connections. The latter is a more interesting choice, because it allows reconfiguring of the implemented functionality. This is especially useful in the development and testing phase, where lots of changes to the design are expected to happen, or when one needs only a small number of chips, where the expenses of manufacturing an ASIC would be out of proportion. On the other hand, the design of an ASIC provides greater control over the hardware features, like including analogue and digital blocks on the same ASIC, which may not be easily added to an FPGA. Another example in the context of collider experiments is the required radiation hardness of the hardware inside the detector not fulfilled by the FPGAs. Also, due to the additional configuration layer, the same design takes up more space on an FPGA, so for larger projects size may be an issue.

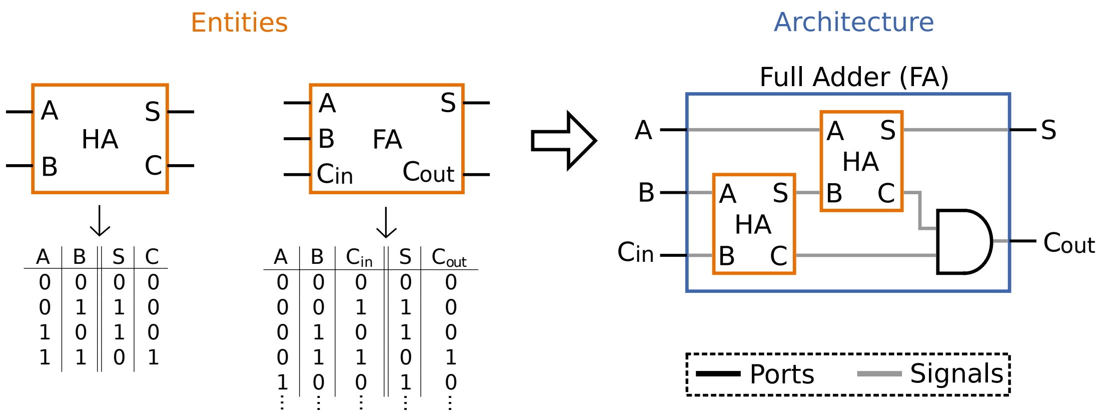
2.4 VHDL applications and Xilinx Vivado workflow

Instead of manually defining and connecting the logic blocks using the schematic design entry, a higher level hardware description language can be used for specifying the FPGAs behaviour. In this project, the Very High Speed Integrated Circuit Hardware Description Language (VHDL) [7] is used. Because structural diagrams are used in Sec. 2.6 and Sec. 5.3, their background is described in the following. An example showcasing the objects described in the text can be seen in Fig. 2.2.

In VHDL, logic is described by defining logical building blocks, called *entities*, with input and outputs called *ports*. The definition of an entity promises certain behaviour on the ports, but does not describe, how this behaviour is achieved. A software analogue are interfaces, that dictate what input and output methods a class has, but do not specify an implementation. The implementation of the entities functionality is called an *architecture*. Within the architecture definitions, multiple strategies may be used to describe it. In one of them, the *structural* description, the architecture of one entity consists of any number of other entities, whose ports are connected by *signals*. This results in a hierarchical design, which can be easily represented graphically, like in Fig. 2.2. The structural description intuitively corresponds to the way one would solve a problem by gradually dividing it into smaller tasks. Another type of architecture is the *behavioural* description. Instead of specifying what components are connected in which way to produce a certain behaviour, it directly describes the behaviour the end result has. As this is a more abstract level of description, the actual components and their connections have to be generated by the design software. On the other hand, this leaves more liberty to the design tool to better target the used technology. [8]

When designing a VHDL system, all these elements are specified in textual format using VHDLs syntax. How the code is then further processed depends on the tooling environment. For this project, the Xilinx Vivado suite is used. One entity, the top level module, is selected to represent the behaviour of the entire FPGA. A behavioural simulation of the code can be used to verify it on a logic level. This does not take into account hardware aspects like timing, and therefore does not guarantee success in the final FPGA. The first step towards the FPGA design is the synthesis of the code. The synthesis generates a list of logic blocks and the connections, called nets, between them from the VHDL hardware description. The implementation step then chooses the locations of the blocks and nets inside the FPGA. This is non-trivial, because beyond the fixed size of the chip constraining the layout, there may be constraints in the time it may take for some logic blocks to produce a result, depending on the application. These timing constraints, together with the related clock definitions and the description of how the hardware pins are connected to the ports of the top level module, are specified in a so-called constraints file. Once the implementation is done, a bitstream containing the implementations information is created, which can be finally used to program the FPGA.

Besides the behavioural simulation, Integrated Logic Analyzers (ILAs) may be used for debugging the project. These are injected into the design after the synthesis step, and connect to selected nets. When running the design on the FPGA, they buffer the values of the nets over time, and write these buffers back to Vivado over the JTAG debugging interface, once a selectable trigger condition is met. This way, one can directly see what is happening inside the FPGA in a given situation. ILAs are also used for testing the programmer software, as described in Sec. 5.2.



A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
⋮	⋮	⋮	⋮	⋮

Figure 2.2: Using a half-adder and a full-adder as an example, to illustrate how entities with ports are implemented by a structural architecture using signals and instances of other entities. In the architecture diagram, the concrete implementation of the half-adder does not need to be known, as long as its functionality is guaranteed by it.

2.5 I2C protocol

The I2C protocol [9] is a two-wire serial bus. The hardware devices on the bus have a numeric address and are assigned a role of either master or slave, with at least one master present on the bus. All devices are attached to the same two serial lines, the serial clock (SCL) and the serial data (SDA) line. Both of these lines are tied to logic high using a pull-up resistor. The SCL line is controlled by the masters, and the SDA line is controlled by the device currently writing data to the bus, as uniquely specified by the protocol. Only the masters may initiate a communication.

The bytes are transmitted bit-by-bit, grouped into single bytes and delimited by the START, STOP, ACK and NACK signals. A transmission is always started by a START and ended by a STOP condition from the master. When using 7-bit addressing, the first byte is also always the address of the slave, with which communication shall be established, and a read/write flag, indicating the direction of the transmission. This way, even with multiple slaves connected to the same data lines, a single slave may be selected for communication. After each transmitted byte, the receiving device has to report status by giving either an acknowledge (ACK) or not acknowledge (NACK) signal, where the not acknowledge signal reports an error, ending the transmission. Giving no signal automatically leads to a not acknowledge. Thus, the bus can be scanned for the connected slaves by initiating a transfer using the START signal and the first byte, and then listening, whether the addressed slave responds with ACK, or a NACK if no slave with the given address is present. A diagram showing the values of SDA and SCL during a transmission can be seen in Fig. 2.3.

The I2C protocol is only a framework on how the bytes are transmitted between master and slave. How they are interpreted depends on the device.

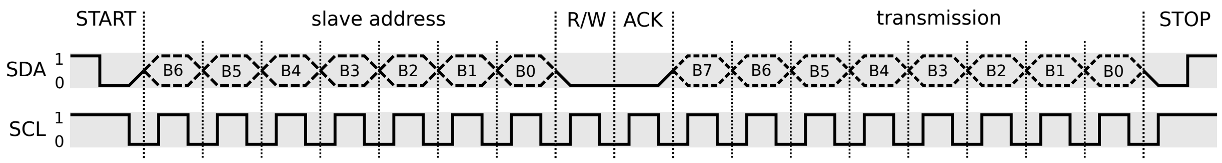


Figure 2.3: An example one-byte write operation (R/W = 0) in the I2C protocol. The addressed slave controls the SDA line only during the acknowledge (ACK).

2.6 The lpGBT

The lpGBT is a transceiver ASIC for use in collider experiments. The context in which it is used is already described in Sec. 2.2.2. Its key features are radiation hardness and high data rates, as required by operation inside the detector. It is robust against radiation damage, which is achieved using 65 nm CMOS technology. Also, special measures for countering the SEUs are taken, for example by triplicating memory cells and using an error correction scheme in the transmission.

The lpGBT is designed to be flexible, to allow use in multiple different environments. This includes, for example, selectable data rates for the optical link and the E-Ports connecting to the front-end modules, the chooseable data directions (transmitter, receiver, transceiver), the hardware pin configurations, and settings regarding the modules processing the communication in general. It also contains debugging features, like a test pattern generator, and experiment control and monitoring features, like the three I2C master modules and the SEU counter.

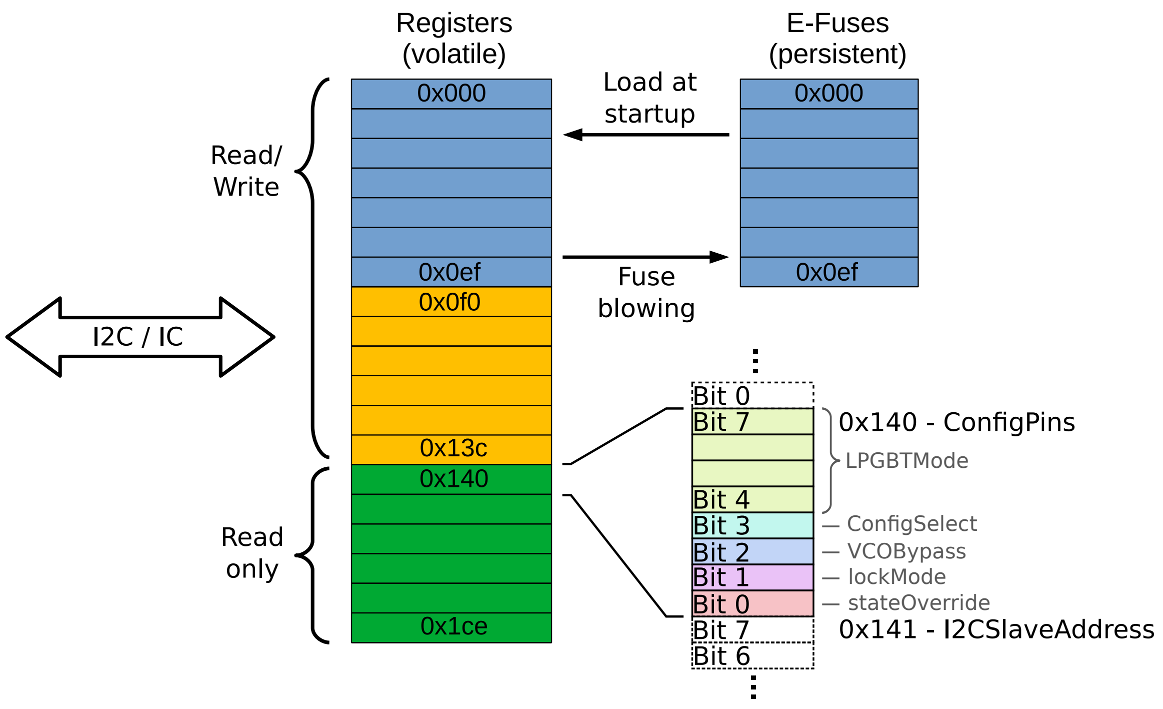


Figure 2.4: The register configuration of the lpGBT. The first part of the physical registers is duplicated into E-Fuses. The physical register `0x140` is shown in a detail view, revealing how it is composed of logical registers, spanning a varying number of bits.

All these features are controlled through registers, an overview of which can be seen in Fig. 2.4. The physical registers are one byte memory cells with addresses ranging from `0x000` to `0x1ce`. Within them, logical registers containing the actual configuration are placed. Logical registers may take up single bits within a physical register, or they may also span over multiple physical registers. The way the logical registers are interpreted varies, and their exact usage is specified in the lpGBTs manual [12].

The physical register values can be read and written either internally by the internal control (IC) field inside the data frames sent via the optical link, or externally via an I2C slave interface on the lpGBT. Some registers only allow read access, as they report the lpGBTs status or provide information not changeable over these two channels, like pin configuration readbacks. A register configuration is mandatory for operation. The registers critical to startup and operation are duplicated into E-Fuses, which may be blown to permanently store a configuration on the chip. This way, instead of having to manually configure the chip at each startup, the configuration can be automatically loaded from the E-Fuses. The fuse blowing is performed through write operations to designated registers, while a fusing voltage is applied to the respective pin.

According to the lpGBT specifications, when using the I2C protocol, the registers can be set by writing the lower and upper byte of the register address to be written to, and then writing the contents. The contents may be one or more bytes, and the register address is automatically incremented after each byte. A register read is initiated by writing the register address. Then, instead of issuing a STOP condition to end the communication, a START condition is repeated, after which the bytes can be polled from the lpGBT, until the master gives a NACK. Again, the address increments automatically.

3 Hardware environment

3.1 lpGBT emulator

An alternative to the lpGBT ASIC is the lpGBT FPGA emulator [13]. The lpGBT emulators top level module structure can be seen in Fig. 2.4. It includes modules for generating the clock signals, the E-Port EPortRx and EPortTx data signals and for serialising the data. These are controlled by the register module, as expected. The two configuration paths are implemented in their own modules controlling the register module, the I2C slave module allowing external configuration over the I2C bus, and the IC interface, which uses the data from the high speed optical link.

A Xilinx Ultrascale+ KCU116 development board is used to host the lpGBT emulator. The emulator is not limited to just this board, although an alternative board must provide a sufficient transceiver performance of at least 10.24 Gb/s, and the constraints file would have to be adapted for it.

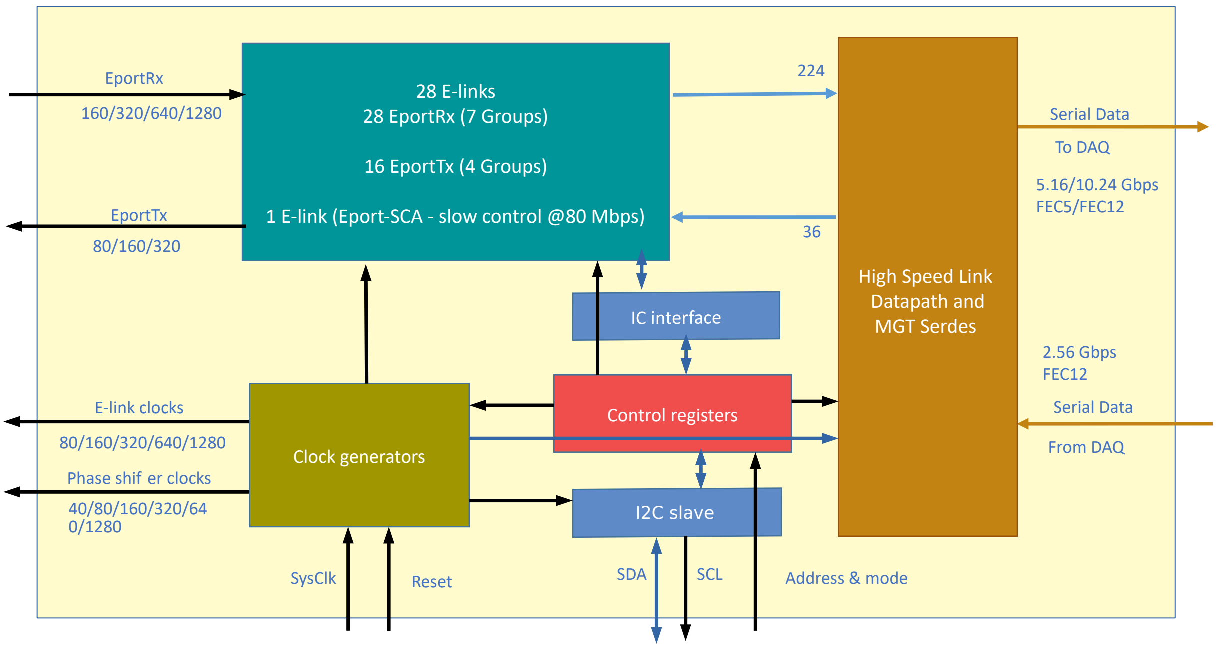


Figure 3.1: The top level module view on the lpGBT FPGA emulator [14].

3.2 CERN USB-I2C dongle

Because the lpGBT programmer software targets conventional computers, which do not provide an I2C interface, a bridge between the computers I/O ports and the I2C bus is needed. The CERN USB-I2C dongle is hardware developed for configuring the GBTx transceiver ASIC [15], the lpGBTs predecessor. It can be connected to the computer running the lpGBT programmer software on one of the USB ports, and on the other side control one or more lpGBT instances as a master on the I2C bus. An Atmel ATmega88PA microcontroller with built-in support for the I2C protocol is used in the dongle hardware for this task.

The microcontroller accepts a USB command byte array of fixed length from the programmer, performs the action specified by the first byte of the command array, and sends a result byte array of the same length back. The arguments expected in the input and produced as the output depend on the command. All commands used by the programmer software are summarised in Tab. 3.1.

Because it was developed for the GBTx, it already contains features specific to this use case. For one, it contains two open drain reset ports for resetting a connected lpGBT, or other devices using active low reset pins. Additionally, a 2 ms long 1.5 V pulse and a 3.3 V fusing voltage on respective pins are controllable. They are used in the fuse burning process of the GBTx, and may be partially reused for blowing the lpGBTs fuses. The dongle also provides an adjustable target supply voltage, which can be switched on and off using the USB commands. Just for configuring the lpGBT, no modifications to the dongle firmware or hardware with respect to the original GBTx version are necessary. However, before blowing the fuses, the 3.3 V fusing voltage has to be reduced to 2.5 V, to be compatible with the lpGBT.

In Fig. 3.2, the hardware setup for configuring one or more lpGBTs can be seen. Besides the SDA and SCL lines of the I2C bus connected to all lpGBTs, a reset chain and the connection of the fusing voltage is shown. The reset chain works by connecting one open drain output of the dongle to the first lpGBTs reset port, and then carrying the lpGBTs reset output into the next lpGBTs reset input. A reset signal from the dongle is then distributed along the chain. The behaviour of the reset output can also be configured through the lpGBTs registers. The fusing voltage connection is only needed for blowing the fuses.

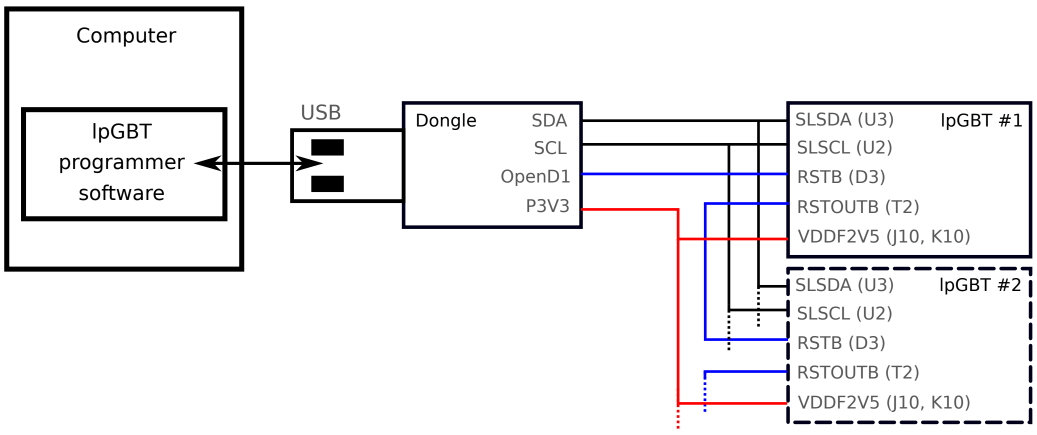


Figure 3.2: The hardware setup of the lpGBT programmer softwares target use case. A reset chain is shown in blue, and the fusing voltage connection is shown in red. The coloured connections are optional, if only a configuration of the chip shall be performed.

Command	Code	Arguments	Output	Description
`I2CTRANS`	1	$1 + N$ bytes: address + transmission data	M bytes: received data	Write data to the slave with the given address, repeat the START command, and return the response
`I2CWRITE`	2	$1 + N$ bytes: address + transmission data	-	Write data to the slave with the given address
`I2CSCAN`	4	2 bytes: start adr., end adr.	N bytes: addresses of the detected slaves	Scan the bus for slaves in the given address range using the technique described in Sec. 2.5
`VERSIONGET`	100	-	2 bytes: dongle version	Retrieve the dongle version
`GOBOOTLOAD`	107	-	-	Switch the dongle to bootloader mode
`VTARGETSET`	200	1 byte: 1/0	-	Switch the target supply voltage on/off
`IO1SET`	203	1 byte: 1/0	-	Switch the open drain reset port 1 on/off
`IO2SET`	205	1 byte: 1/0	-	Switch the open drain reset port 2 on/off
`VFUSESET`	207	1 byte: 1/0	-	Switch the fusing voltage on/off

Table 3.1: Overview of the dongle commands used in the programmer software. The code field refers to the value of the first byte of the USB argument array.

4 Software development

4.1 Motivation and requirements

As already mentioned, configuring the lpGBT is necessary for its operation. The final goal is to develop an application for doing this, but some choices have to be made along the way. A discussion of the choices, presented in the following, motivates the programmer softwares current specifications.

The first choice is, whether to use the I2C or the IC configuration path. Since the IC configuration path uses the optical link, it already requires a working configuration within the registers, meaning it has to be configured at startup using either the I2C protocol or the fuses. In reverse, the fuses cannot be blown using the IC path without an external I2C programmer, as this also requires access to the registers. Therefore, the more flexible choice is the I2C configuration path. Additionally, I2C configuration is easier to set up, when only the lpGBT ASIC or emulator without the other components of the readout chain are available.

Secondly, the CERN USB-I2C dongle is chosen as the device controlling the I2C bus, so that the already existing dongle hardware already available at the involved institutes can be reused. This is in contrast to the piGBT lpGBT configuration software [16] developed before the lpGBT programmer software, as it uses a Raspberry Pi single-board computer in place of the computer/dongle combination, which is not available in some institutes.

The programmer software targets manual configuration by a person, instead of exposing the configuration functionalities to a scripting environment for automatic configuration. Thus, a GUI is chosen as the interface between the user and the software, as opposed to an application programming interface (API) or a command line interface (CLI).

The "GBTx programmer" is the GUI configuration program of the GBTx [17]. It also uses the dongle to configure the GBTx over the I2C interface, similar to what this project achieves. Therefore, as an additional benefit of using the I2C configuration path and the dongle, parts of the graphical interface design and the code for controlling the dongle could be adapted to this project from the GBTx programmer software.

This adaptation requires the use of the Java programming language, in which the GBTx programmer software is written. Its benefits with regards to this project are the included Swing GUI framework and the cross-platform support. A disadvantage is the dependency on the Java Runtime Environment (JRE) for executing the programmer software on the target computer. The additional JRE layer also makes it harder to add an API for a scripting language like Python besides the GUI, even if the class structure of the programmer software allows it. For the GBTx, this is addressed by having two separate implementations: the Java GBTx programmer GUI and a smaller command line and scripting module written in Python.

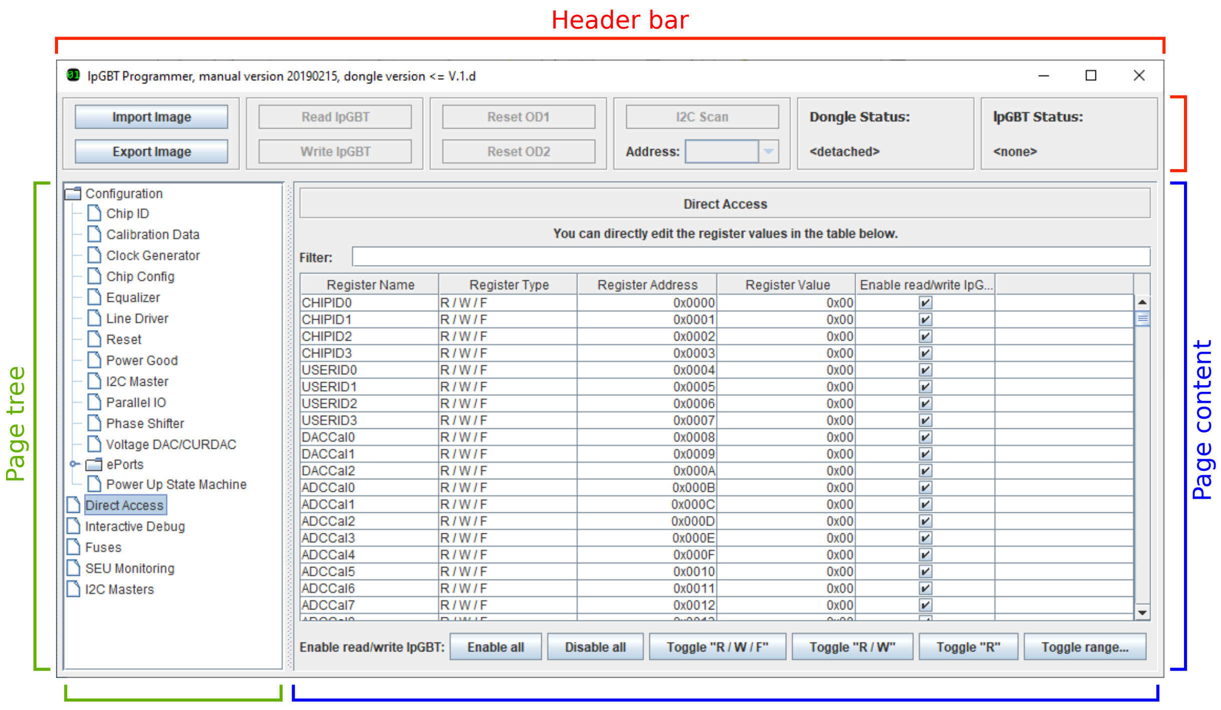
With these choices, a set of tasks the programmer software should be able to perform are defined, going from direct hardware control to tasks using higher abstraction: The programmer software should, on a very basic level, be able to control the dongle functionalities by sending the USB commands listed in Tab. 3.1. Using the I2C dongle commands, the physical register byte values should be presented, with the option to read them from or write them to the lpGBT. Since both the dongle and the I2C bus support multiple slaves, the lpGBT programmer software should also be able to scan and select from the devices on the bus. In addition to the physical register values, the logical register values should be selectable, too, as these are more relevant to the configuration itself. Due to the different interpretations of the logical registers bit patterns by the lpGBT, this is the most complex task. On a similar note, higher level access should be provided to the other lpGBT functionalities controlled through the registers, like the fuse blowing process. Especially the higher level interfaces for the logical registers and the lpGBT functionalities can make good use of the GUI concept, since a GUI better supports, and in some cases requires, a more user-friendly and less technical presentation than for example a command line interface.

The following sections describe, how these tasks can be performed in the project implementation.

4.2 General user interface

A screenshot of the lpGBT programmer GUI is shown in Fig. 4.1. Generally, the window is divided into the header bar, the page tree, and the page content. The pages, selectable in the page tree and displayed in the page content, provide a view onto the workspace lpGBT instance, which is a replica of the lpGBTs registers in the computers memory. The header bar then controls, how the workspace values are processed. Also, hardware related actions can be performed in the header.

Going from left to right in the screenshot of Fig. 4.1, the first controls of the header



Register Name	Register Type	Register Address	Register Value	Enable read/write lpG...
CHIPID0	R / W / F	0x0000	0x00	✓
CHIPID1	R / W / F	0x0001	0x00	✓
CHIPID2	R / W / F	0x0002	0x00	✓
CHIPID3	R / W / F	0x0003	0x00	✓
USERID0	R / W / F	0x0004	0x00	✓
USERID1	R / W / F	0x0005	0x00	✓
USERID2	R / W / F	0x0006	0x00	✓
USERID3	R / W / F	0x0007	0x00	✓
DACCal0	R / W / F	0x0008	0x00	✓
DACCal1	R / W / F	0x0009	0x00	✓
DACCal2	R / W / F	0x000A	0x00	✓
ADCCal0	R / W / F	0x000B	0x00	✓
ADCCal1	R / W / F	0x000C	0x00	✓
ADCCal2	R / W / F	0x000D	0x00	✓
ADCCal3	R / W / F	0x000E	0x00	✓
ADCCal4	R / W / F	0x000F	0x00	✓
ADCCal5	R / W / F	0x0010	0x00	✓
ADCCal6	R / W / F	0x0011	0x00	✓
ADCCal7	R / W / F	0x0012	0x00	✓

Figure 4.1: An annotated screenshot of the lpGBT programmer GUI, showing the three main areas of the user interface.

are the buttons for writing selectable parts or all of the physical registers to an image file, and in turn reading the image files into the workspace instance. After that follow the two buttons for writing the workspace values to the selected lpGBT, or reading the selected lpGBTs register values. By default, all registers are read/written, although the user can select which registers are read or written on a per-register basis, by using the page described in Sec. 4.4. The header continues with two reset buttons. Upon activation, they send a 500 ms reset pulse to the respective open drain reset pins of the dongle. A fourth group of controls is used to scan the I2C bus for devices, and to set the target lpGBT. The selection is done by choosing the lpGBTs slave address in the "Address" combo box. Its selectable values are updated at startup, or each time an I2C scan is performed. Lastly, two labels show the status of the dongle and the lpGBT, if they can be accessed by the programmer software. For the dongle, the firmware version, as reported by the `VERSIONGET` USB command (see Tab. 3.1), is shown. For the lpGBT, the first four register values specifying the chips ID are read and displayed on success. Each button of the header is only enabled for use, when the necessary hardware is detected by the programmer. For example, when taking the screenshot of Fig. 4.1, no dongle was connected and only the lpGBT image buttons are enabled.

All register configuration is performed on the workspace instance, and has to be actively written to the lpGBT to take effect. This is not necessarily the case for the additional interfaces described in Sec. 4.6. The decoupling from the hardware allows editing lpGBT image files without having a dongle or lpGBT attached to the computer.

The view onto the workspace values is paginated, because the distinction between physical and logical registers creates different ways of representing the same register values graphically. Also, the large number of registers (459) requires the pagination for accessibility.

4.3 Controlling the dongle

The dongle USB commands themselves were already given in Tab. 3.1, but some comments are due, on how they are being used. On initialisation of a dongle found on the USB bus, its target supply voltage is enabled and its fuse voltage is disabled, using the `VTARGETSET` and `VFUSESET` commands. The `IO1SET`, `IO2SET`, `VERSIONGET` and `GOBOOTLOAD` commands are used directly, without additional logic behind it.

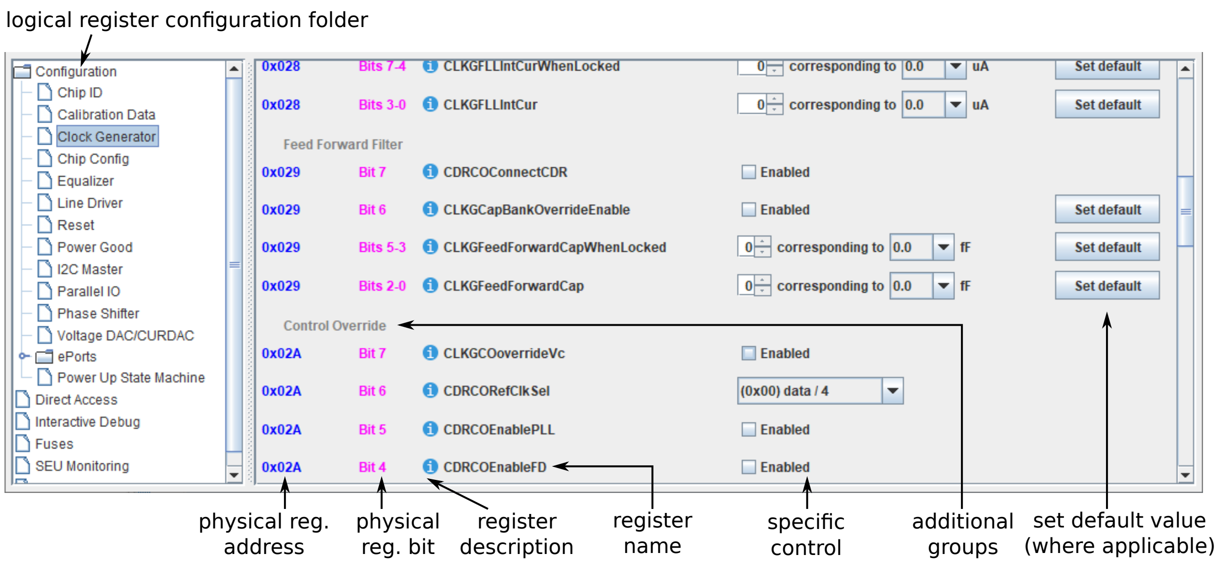
This is not the case for `I2CTRANS` and `I2CWRITE`, as the number of registers, that may be read or written in a single given dongle command, is limited by the USB command byte arrays length, which depends on the firmware version of the dongle. To speed up the communication, the goal is to read/write the maximum possible number of register the dongle allows with each command, since for each new command the lpGBTs slave address and the register address needs to be repeated. On the other hand, registers disabled for read/write and gaps in the registers (address range `0x13d - 0x13f`) shall be avoided. This problem is solved by first dividing the registers into ranges to be read/written individually in the programmer code. A range is found by starting from a start address, and incrementing it, until the start address is not blocked. Then, the frame is expanded until either the maximum length is reached, or another blocked register is hit. The start address of the first range is `0x000`, and the start of each new range is the first register after the previous range. For the write operations, this process is iterated until the read-only registers are reached, and for the read operations, this process is iterated until the end of the physical registers. A similar strategy is used for scanning the I2C bus, although in this case the address range to be scanned (`0x01 - 0x7f`) is continuous. In agreement with the lpGBTs I2C interface, the `I2CTRANS` command is used for reading, and the `I2CWRITE` command is used for writing the registers.

Even if technically possible, switching between multiple dongles connected to the PC is not supported. When more than one dongle is connected, the first one to be reported by the USB library is used.

4.4 Interface of the physical registers

The interface of the physical registers is already shown in Fig. 4.1. It contains a table of values, where each row displays one physical register of the workspace instance. Within a row, the name of the register as specified in the manual, its type, address and value are presented. The type differentiates between the register regions. It reads "R/W/F" for fuseable registers, "R/W" for writable, but not fuseable registers, and "R" for the read-only registers. The registers can be filtered by name. Also, each register can be enabled or disabled for read/write when using the header buttons. For convenience, buttons for grouped control over this read/write enable feature are placed underneath the table. The actual editing of the values is done by clicking the value cells of writable registers. After the click, the cell turns into an editor, and the hexadecimal byte value can be typed in or selected using the spinner controls.

4.5 Interface of the logical registers



0x028	Bits 7-4	CLKGFLLIntCurWhenLocked	0 corresponding to 0.0 uA	Set default
0x028	Bits 3-0	CLKGFLLIntCur	0 corresponding to 0.0 uA	Set default
Feed Forward Filter				
0x029	Bit 7	CDRCOConnectCDR	Enabled	
0x029	Bit 6	CLKGCapBankOverrideEnable	Enabled	Set default
0x029	Bits 5-3	CLKGFeedForwardCapWhenLocked	0 corresponding to 0.0 fF	Set default
0x029	Bits 2-0	CLKGFeedForwardCap	0 corresponding to 0.0 fF	Set default
Control Override				
0x02A	Bit 7	CLKGCOoverrideVc	Enabled	
0x02A	Bit 6	CDRCORefClkSel	(0x00) data / 4	
0x02A	Bit 5	CDRCOEnablePLL	Enabled	
0x02A	Bit 4	CDRCOEnableFD	Enabled	

Figure 4.2: Annotated screenshot of a page for configuring the logical registers.

The logical register configuration pages, collected inside the "Configuration" folder of the tree, which can be seen in Fig. 4.2, provide a list-like view onto the logical registers. Each line consists of a description of the placement within the physical registers, a description of the registers purpose, the name, a register-specific control and, where applicable, a button for resetting it to its default value. The placement part specifies the physical register addresses the logical register is contained in, and in the case of it being just one physical register, also which bits of it are mapped. Hovering the mouse over the information icon besides the name reveals the registers description as a tool tip. In addition to the placement inside different pages within the tree, groups within the pages can be defined, as shown in the figure. The pages of the tree are oriented towards how the registers are grouped in the lpGBTs manual.

The lines are generated just from a description of the logical registers. This description also assigns the logical registers one of currently eight types. The type specifies, how the register value is interpreted in the lpGBT, and consequentially, what control is chosen to set its values in the GUI. This can be seen in Fig. 4.2, where the lines have different register-specific controls. An overview on what controls are available is given in Fig. 4.3.

Setting the value through the control immediately changes the corresponding bits inside the workspace instance, which can be verified in the physical register page. The other bits of the physical register, that do not belong to the modified logical register, are left unchanged. This is one of the biggest strengths of the programmer software, as the required modifications on the bit level, which are error prone when done manually, are automatically performed. Again, only the workspace instance, instead of the actual lpGBT device configuration, is changed. One should be aware, that trying to write logical registers included in physical registers disabled for read/write will for that reason not succeed.

Only the fuseable registers are implemented in the GUI using this type of page. The reason is, that the non-fuseable registers are less static, in that they control live configuration, monitoring and experiment control features. Controlling them would certainly be possible with this page type, but it is not the optimal way to represent these registers graphically and for user interaction. As an example, the lpGBTs I2C masters are controlled through writing encoded commands into logical registers spanning at least 4 physical registers. One I2C master transaction requires sending up to 6 of such commands. Thus, controlling the I2C masters by hand using this kind of page would be very tedious, and special pages addressing these features are required in the GUI. The pages addressing such features implemented at the time of writing are described in Sec. 4.6.

4.6 Additional interfaces

Other lpGBT modules controlled by the registers are accessed through their own customised pages.

The "Fuses" page controls the fuse blowing process. It gives an overview over the register contents of the fuseable registers, like the page described in Sec. 4.4. In addition,

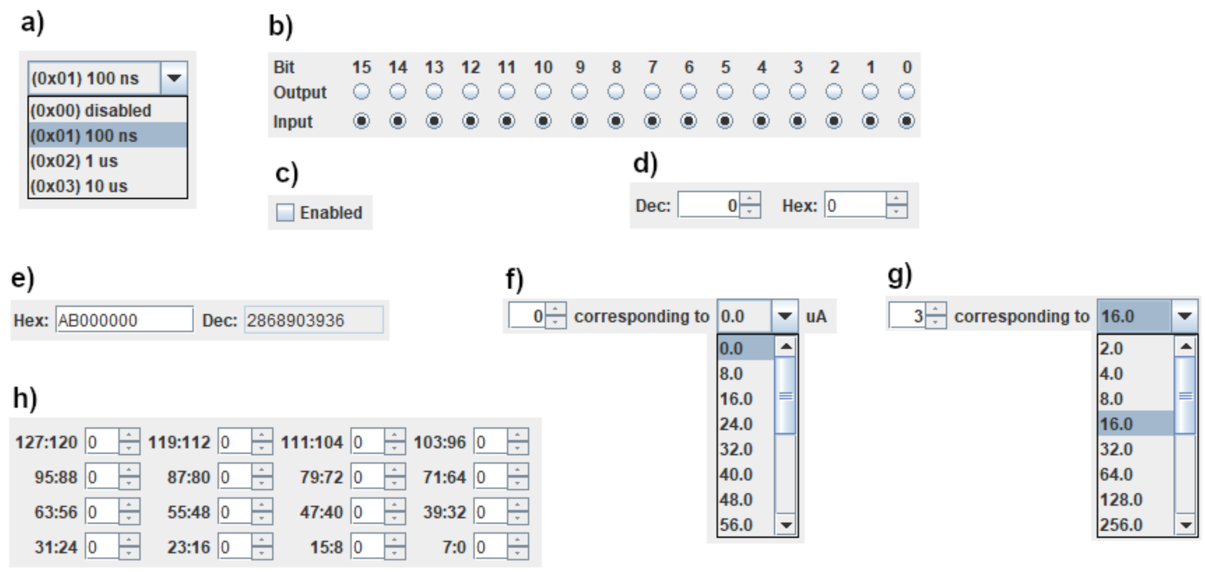


Figure 4.3: The available controls for setting the logical register values. For further information, please see the programmers manual. **a)** (*AssocListSlice*) free assignment between value and quantity **b)** (*BitChoiceSlice*) choice on a by-bit basis **c)** (*BooleanSlice*) a single bit flag **d)** (*NumericSlice*) the register value is interpreted as an unsigned integer **e)** (*MultiByteSlice*) the register value is interpreted as a 4 byte unsigned integer **f)** (*IntervalSlice*) the value selects a quantity within an interval **g)** (*CalculatingSlice*) the value maps to the quantity through a function **h)** (*MultiByteSlice*) a sequence of bytes without further interpretation

when the fuses are already blown, their values can be read and then displayed besides or compared to the workspace instance values. Blowing the fuses to the displayed workspace register values is also possible. Before blowing the fuses, the programmer software asserts that the `updateEnable`, `pllConfigDone` and `dllConfigDone` flags are enabled, if the corresponding check boxes are set. These flags must be set, in order to operate the lpGBT at startup using the configuration stored in the fuses.

To pick up the example of Sec. 4.5, a separate page is implemented for the lpGBTs I2C masters. Within, the masters configuration (master select, frequency, pin configuration) can be set and bytes can be read and written to the connected slaves. The read and write operations are specified textually in a script format. Each line in the script starts with a command, followed by the commands arguments separated by whitespace. The script is then interpreted from start to end, and the commands are executed sequentially. The most important commands are the `read` command, which takes the number of bytes to be read from the slave as an argument, and the `write` command, which writes the argument bytes to the slave. All other commands are used to change the master settings during execution of the script. When the script is run, an output text containing the slaves response, notifications of changed settings and error messages is generated. A more detailed description of the script syntax is given in the prepared programmers manual. The benefit of the script format, compared to special controls for reading/writing the values, is that initialisation procedures of one or more devices consisting of multiple write operations, also across the different masters, can be performed using just one script. Also, this script can be saved to and loaded from the filesystem, and does not have to be manually input every time, which would have been the case for the special controls.

Another page is the "SEU Monitoring" page. Here, the lpGBTs internal SEU counter can be logged and displayed over time. The log files name, the rate at which the value is read and the data width of the pages graph can be set. This feature was inspired by a similar tab in the GBTx programmer GUI.

For debugging purposes, an "Interactive Debug" page hidden behind the `--debug` command line flag was created. It contains controls for writing and reading single physical registers, as well as controls for performing the testing procedure described in Sec. 5.2.

4.7 Class structure

There are too many classes to discuss all in detail, but the class structure of the most important classes and why it was chosen is presented in the following. Of course, this section refers to the class structure at the time of writing. In the hopes of making it easier to understand, an illustration of the text is shown in Fig. 4.4. In the first instance, the project is split into the model classes, taking care of the logic and hardware control, on one side and the GUI classes on the other side. The model classes do not refer to the GUI classes, so the possibility of reusing them independently in a different interface, such as a command line interface, is left open.

The main class is `MainWindow`. It creates a window including the layout and functionality of the header panel and the page tree. The header controls the hardware, so `MainWindow` creates an instance of the `HardwareManager`, which is supposed to be the one class for connecting the hardware classes to the GUI. It contains the two main hardware related classes, the `Dongle` class adapted from the GBTx programmer software, and the `LpGBT` class containing the workspace instance, the fuse values and the currently targeted I2C address. Access to the `HardwareManager` and `LpGBT` is only given through the interfaces `IHardwareManager` and `ILpGBT`. During development, this enabled connecting the former `DummyHardwareManager` and `DummyLpGBT` test classes implementing these interfaces to the GUI instead. So the GUI could be tested without a working hardware implementation, but later it could still be easily changed to use the real hardware classes, by just changing the code instantiating the two objects.

For implementing the application pages, the abstract `FeaturePanel` class is created. It provides the methods needed to display the child class objects in the tree. Also, it contains event methods called by the `MainWindow` on page initialisation, select, deselect and for refreshing the displayed data on change. A new tree page is therefore created simply by implementing these predefined methods in a child class extending `FeaturePanel`, and adding it to the tree inside the `MainWindow`. As a result of this design, the pages are decoupled from one another, and the implementation of the different configuration functionalities are automatically placed inside their own classes.

The `SlicePanel` shown in Sec. 4.5 for configuring the logical registers, called slices in the implementation, requires the most additional structure. The custom controls in each line are instances of the classes extending `SliceDisplayBase`. They are for each page generated from a `SliceBase` array describing the logical registers, which are stored in the static class `RegisterSlices`. The conversion between the model classes extending `SliceBase` and the GUI classes extending `SliceDisplayBase` is done in the `SliceDisplayCreator` by using the visitor pattern [10]. This free assignment between the model and GUI classes allows one model to have two different GUI controls (`MultiByteSlice`), or two models to have the same control (`BooleanSliceDisplay`). Inside the model classes, `BitRange` and `BitString` are used to map the logical register values to the physical register bits.

The benefit of using the auto-generated pages is that changes and bug fixes in the model or GUI classes are automatically reflected in all instances throughout the pages. Also, the layout of the pages can be changed in one single place. The GUI generation just from the small `SliceBase` objects inside `RegisterSlices` allows quick development and modification of the GUI, as more or less required by the number of ≈ 200 logical registers currently implemented. A downside to this design is the loss of flexibility in the GUI: If one wants to deviate from the list-like structure, an entirely new page has to be implemented.

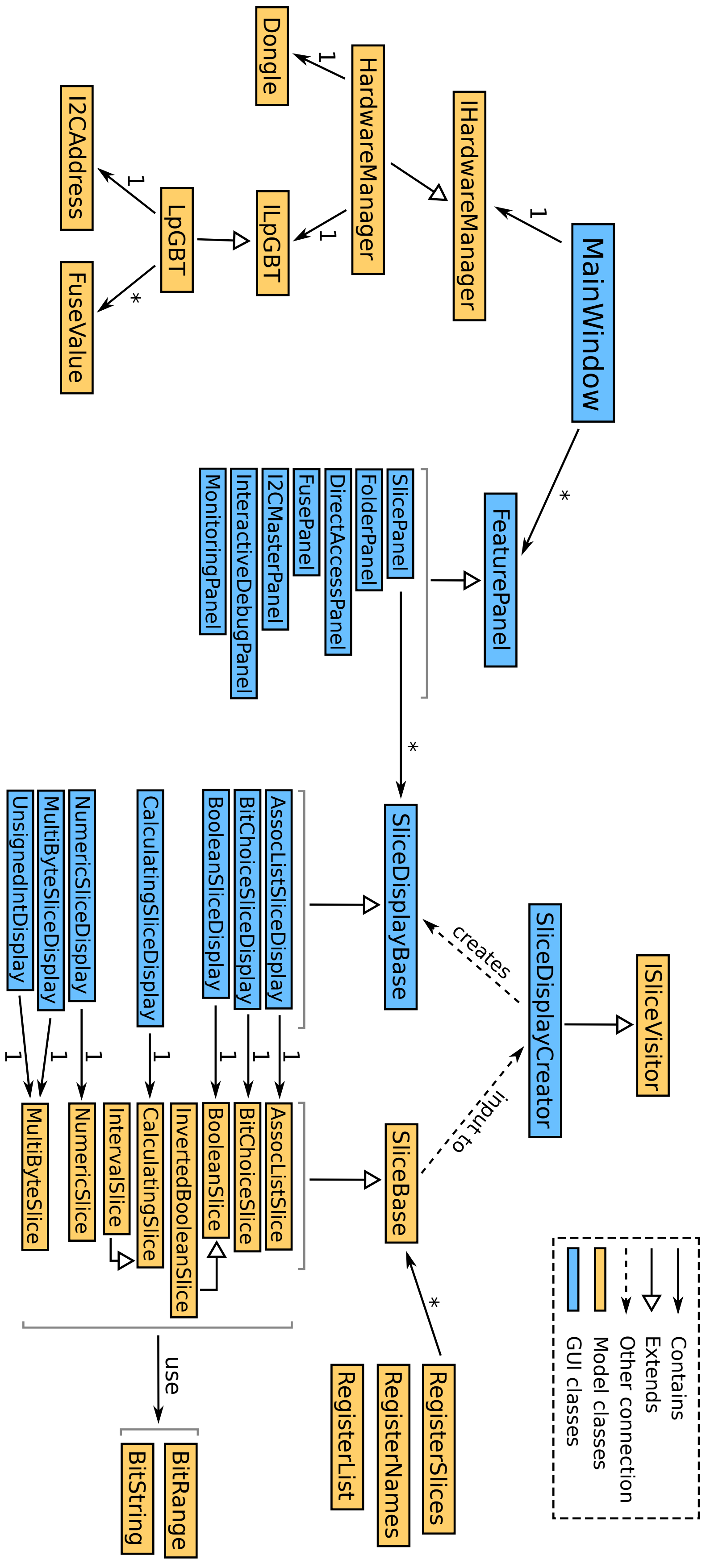


Figure 4.4: An overview over the class structure of the lpGBT programmer software. This is not a view on all classes, as for example utility classes do not significantly contribute to the main structure of the application and are therefore left out.

5 Testing

5.1 Testing goals and limitations

Beyond testing the user interface during development, the programmer softwares core task of configuring the lpGBT needs to be tested. Ideally, the lpGBT ASIC and a working readout chain are used for the tests. However, the lpGBT ASIC was not available in the testing phase, so it is substituted by the lpGBT emulator. The KCU116 development board hosting the emulator was also neither connected to front-end modules nor a data acquisition (DAQ) system, which are needed for a complete readout chain. So a real-world test case involving configuring and modifying the readout chain features on-the-fly was not possible. Thus, the overall testing possibilities are limited compared to the full setup, as used in the detector. Still, most of the programmer software functionalities can be validated with the lpGBT FPGA implementation, with the exception of the analogue features of the lpGBT and the fuse blowing.

The abstract task of configuring the lpGBT is already broken down into a list of smaller tasks the programmer software can perform in Sec. 4.1. They are also already listed in the order they have to be tested, going from direct hardware control to the higher level features, as hardware control is required for successfully reading and writing the registers, which is in turn required for configuring the other modules of the lpGBT through the higher level interfaces. Consequentially, the whole testing process is divided into two consecutive steps: In the first step, the dongle connection is established, and it is verified, that the programmer software can read and write the lpGBT emulators register values. The performed actions and their results are detailed in Sec. 5.2. The second step tests the hardware setup, in which multiple lpGBT instances are configured over the same bus. For the programmer software this means testing its I2C scan feature and the ability to select between the slaves on the bus. How this is achieved using only one single FPGA development board is shown in Sec. 5.3.

The one task defined in Sec. 4.1, but not addressed by these two testing steps, are the higher level interface implementations of Sec. 4.6. Since the corresponding modules are not, or in the case of the fuses can not, be implemented in the lpGBT emulator (see

Fig. 3.1), the interfaces ability to control the lpGBT ASIC can not be verified with the hardware test setup. The fuse blowing is an irreversible process, so to prevent the untested fuse blowing code from running and possibly permanently misconfiguring an lpGBT ASIC, it is commented out. While having untested features makes the concerned pages useless in practice, their interfaces are working, and the commented out sections of the fuse panel, or the communication code of the other pages, provides a demo implementation, showcasing how the programmer softwares class structure may be used to perform the respective tasks. This way, when the features are actually needed, less effort has to be put in to reach a fully functioning version of these pages.

Another point to discuss is, how the successful tests with the lpGBT emulator translate back to the lpGBT ASIC use case. The dongle is the hardware sending the actual signals, and it is now verified to work at least with the GBTx ASIC and the FPGA lpGBT emulator. With this background, it will surely work with the lpGBT ASIC, too. So as long as the dongle is interfaced correctly by the programmer software, and the order and meaning of the bytes confirm to the lpGBTs specifications, both of which are tested with the lpGBT emulator, the programmer software will also correctly program the lpGBT ASICs registers. After establishing successful configuration of the ASICs registers themselves, their influence on the lpGBTs components solely depends on the lpGBT, and not the programmer software.

5.2 Communication tests

Due to the dongle code being mostly reused from the GBTx programmer software, communication with the dongle, like querying its firmware version, was established quickly. After that, there are three ways to detect changes in the register values, all of which were used to verify the programmer softwares communication with the lpGBT emulator. The two direct ways are reading the register values through the ILAs on the FPGA, or through the I2C slave. Indirectly, changes in the register values may be detected by measuring changed behaviour of the lpGBT emulator.

By tracing the ILA values of the SDA, SCL and register signals over time, one has direct verification of a transactions validity and success. An example of this is shown in Fig. 5.1. The ILAs are great for probing individual transactions, but become tedious when checking the validity for multiple different transactions. This can be automated for the writable registers, by using the second way of reading the registers, the regular I2C path. Ideally, if one would check whether all writable registers read back the correct value after writing all possible values, the testing would be complete. To save time, the range

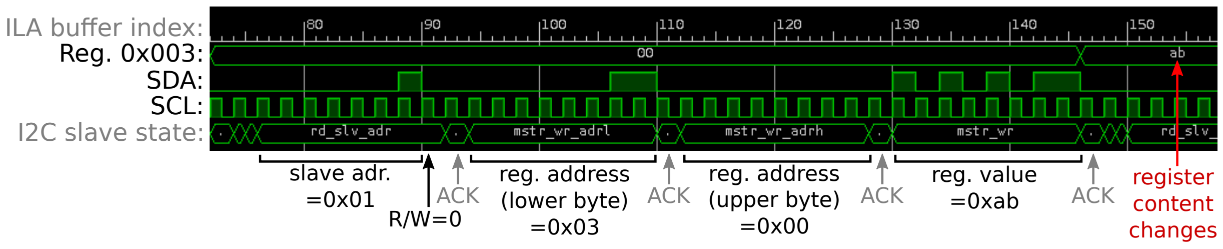


Figure 5.1: A screenshot of the Vivado ILA view, displaying a write transaction. After the write, the register value of the target register changes respectively. The presentation is the same as in Fig. 2.3. The START and STOP conditions cannot be seen directly, because the ILA buffer is set to only capture on SCL change, but they can be indirectly seen in the slave state change.

of all possible values is only sampled using fewer random values. The resulting testing procedure is

1. Set the workspace values to random byte values from `0x00` to `0xFF`, and store a copy of the generated values
2. Write all registers to the device
3. Reset the workspace values by negating
4. Read all registers from the device
5. Compare the workspace values to the copied values of step 1

It may be iterated to get closer to the ideal case. Between two iterations, the value generated in step 1 is guaranteed to change in the implementation. In step 3, the negation is chosen in favour of just resetting the values to a constant, so that no value is distinguished. When running this procedure using the lpGBT emulator, the list of registers passing all iterations of this procedure exactly matches the list of writable registers implemented in the lpGBT emulator. The unimplemented registers are hard-wired to read `0x00`, so their failure is the expected result.

Because both the logical register pages and the specialised pages internally map to the physical registers, these tests also confirm their ability to read from and write to the device. As an example of measuring modified behaviour, an oscilloscope is connected to one of the E-Clock outputs. Their frequency may be set by using the lpGBTs registers, and upon setting the value in the logical register page, the change in the E-Clocks frequency and polarity can indeed be measured, as seen in Fig. 5.2.

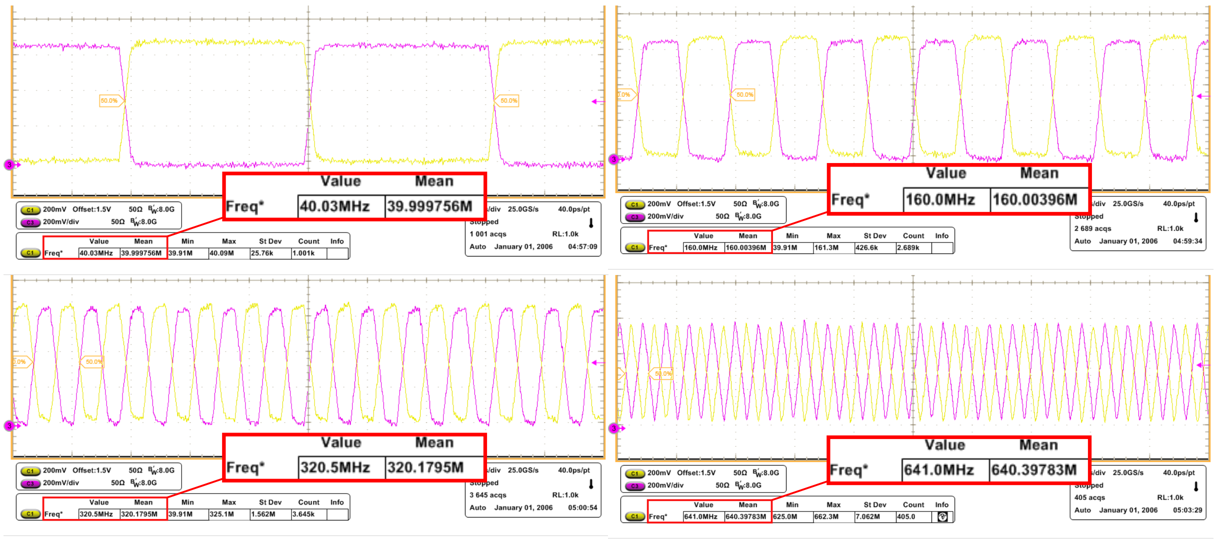


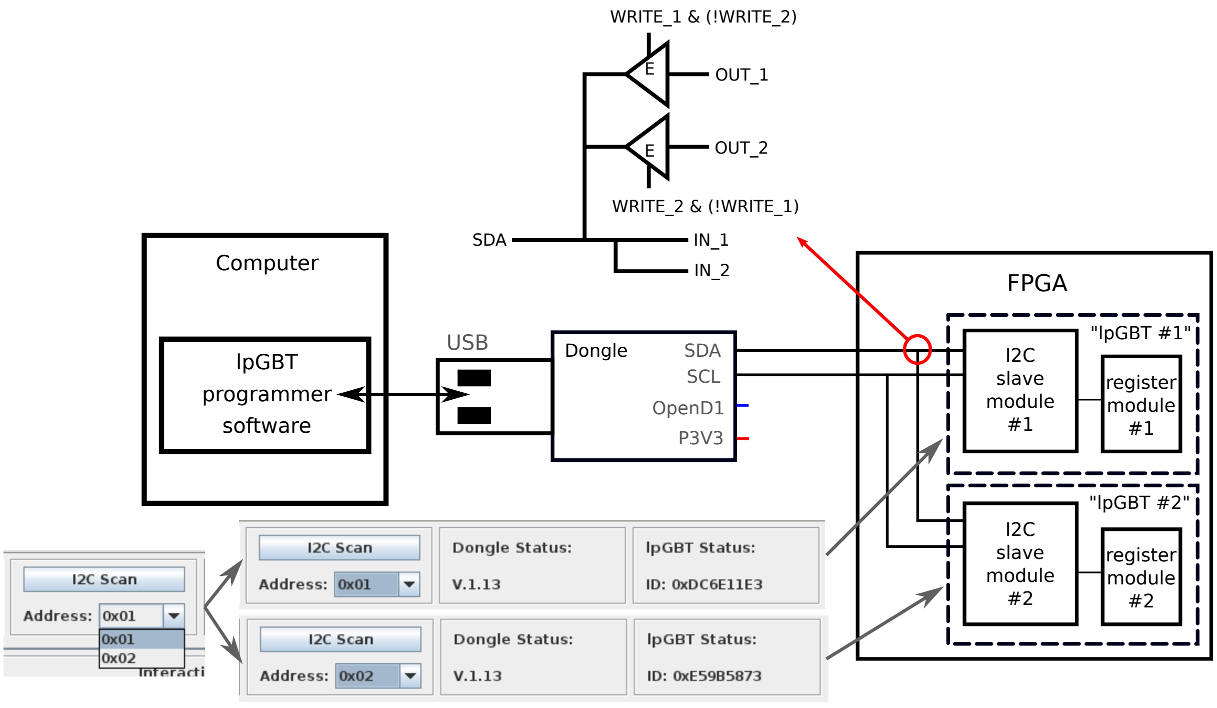
Figure 5.2: A tiled view on the output of a differential E-Clock over time. The frequency changes between the tiles as a result of reconfiguring the settings with the logical register programmer GUI.

5.3 Multi-lpGBT tests

For testing the multi-lpGBT setup with only one FPGA development board, the emulator modules are duplicated inside the FPGA, and thus form two separate lpGBT instances. This is not done with the full emulator, but with a stripped version containing only the necessary modules in each instance, to speed up the implementation step. The stripped emulator consists of the clock generator, I2C slave and register modules. A diagram of the setup parallel to the hardware setup in Fig. 3.2 is shown in Fig. 5.3.

This configuration is handled correctly by the programmer software, too. The two instances are picked up in the I2C scan and are displayed in the combo box for selecting the target. Individually, they still pass the testing procedure described in Sec. 5.2. Also, when switching between the instances one can see, for example in the lpGBT status label or in the physical register values in general, that there indeed exist two separate register instances with different values.

A problem arises when simply trying to connect both SDA lines of the two I2C slave modules, to form the combined SDA port of the FPGA. Because when judging only from the description of the SDA ports on the slave modules, marked in VHDL as both input and output, it is possible that both slave modules actively drive the combined SDA line to different values. This leads to an error in the implementation step, as one net may only have one driver. One possible solution to this problem is to split the single SDA port of the slave module, as present in the original emulator, in three separate ports with distinct directions: An SDA_IN and SDA_OUT port, for the SDA input and output, and the SDA_WRITE port, that specifies which of the two directions is currently used. With this, the SDA_IN ports can be connected to the combined SDA line unconditionally, as they do not drive it. The SDA_OUT ports are connected to the shared line only through tri-state buffers, and only drive the shared line, if mutually excluding conditions are met. This design ensures, that only one slave drives the FPGAs SDA pin at each given time, and it can be therefore implemented successfully.



Address: 0x01 | Dongle Status: V.1.13 | lpGBT Status: ID: 0xDC6E11E3

Address: 0x02 | Dongle Status: V.1.13 | lpGBT Status: ID: 0xE59B5873

Figure 5.3: A diagram showing the structure of the multi-lpGBT test configuration, the results in the GUI and the wiring of the SDA lines as discussed in the text.

6 Conclusion and Perspectives

6.1 Conclusion

As part of its infrastructure, a GUI application for configuring the lpGBT by setting its register values over the I2C protocol was developed. It runs on a conventional computer, but depends on the already existing CERN USB-I2C dongle hardware, for establishing and controlling the I2C bus. Through the GUI, the values of both the physical and the logical registers can be set in separate content pages. Also, experimental pages allow custom tailored control over some of the lpGBTs monitoring and experiment control modules, as well as the fuse reading and blowing process. The programmer softwares ability to read and write the register values was successfully tested with the FPGA lpGBT emulator. This holds true both in the case where one single, full emulator is used in place of the lpGBT ASIC, and in the case where selected modules of the emulator were duplicated to mimic configuration of more than one lpGBT ASIC over the same bus. With these features, the programmer software can be utilised in the setup of a readout system, and in further development of the lpGBT emulator.

6.2 Perspectives

Due to the circumstance of the lpGBT ASIC not being available for testing, the full-featured use case of the programmer software is still partially untested. Further tasks needed for completeness are communication tests with the lpGBT ASIC, although the lpGBT emulator tests are expected to cover them, as discussed in Sec. 5.2. More importantly, tests regarding the higher level interfaces, most critically the page for reading and blowing the fuses, still need to be carried out, along with possible adjustments to the software. On the hardware side, the dongles fusing voltage needs adjustment, to be able to perform the fuse blowing process. Lastly, and with the least priority, there are still lpGBT modules without a corresponding page in the GUI, like the eye opening monitor or the test pattern generator. Implementing them is another possible continuation of the project.

Bibliography

[1] G. Apollinari, et al., *High Luminosity Large Hadron Collider HL-LHC*, CERN Yellow Rep. **(5)**, 1 (2015)

[2] L. Evans, P. Bryant, *LHC Machine*, JINST **3**, S08001 (2008)

[3] ATLAS Collaboration, *The ATLAS Experiment at the CERN Large Hadron Collider*, JINST **3**, S08003 (2008)

[4] ATLAS Collaboration, *ATLAS liquid-argon calorimeter: Technical Design Report*, Technical Report CERN-LHCC-96-041, Geneva (1996)

[5] ATLAS Collaboration, *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*, Technical Report ATLAS-TDR-030, CERN, Geneva (2017)

[6] E. Monteil, et al., *RD53A: a large scale prototype for HL-LHC silicon pixel detector phase 2 upgrades*, PoS **TWEPP2018**, 157 (2019)

[7] *IEEE Std. 1076-2019 - IEEE Standard for VHDL Language Reference Manual* (2019)

[8] C. H. Roth, L. K. John, *Digital Systems Design Using VHDL*, Thomson Learning, 2 edition (2008)

[9] NXP Semiconductors, *I2C-bus specification and user manual* (2014), rev. 6

[10] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional (1994)

Online references

[11] *lpGBT programmer software repository*, URL https://gitlab.cern.ch/askaf/lpgbtprogrammer

[12] lpGBT Design Team, *lpGBT Documentation* (2019), v0.16, URL https://lpgbt.web.cern.ch/lpgbt/v0/

[13] *lpGBT FPGA emulator repository*, URL https://gitlab.cern.ch/askaf/lpgbt-emulator

[14] A. Skaf, *FPGA lpGBT Aggregator/Emulator Status update*, URL https://indico.cern.ch/event/895153/contributions/3798080/attachments/2010917/3359845/LpGBT_Emulator_status.pdf

[15] P. Moreira, J. Christiansen, K. Wyllie, *GBTX MANUAL* (2018), v0.16, URL https://espace.cern.ch/GBT-Project/GBTX/Manuals/gbtxManual.pdf

[16] *piGBT programmer software website*, URL https://pigbt.web.cern.ch/

[17] *GBTX programmer software repository*, URL https://gitlab.cern.ch/gbtproj/gbtxprogrammer

Acknowledgements

Finally, I want to express my gratitude to Dr. Ali Skaf for supporting me over the entire 14 weeks of the thesis in numerous ways, including help in planning the project, answering all my questions, correcting and helping me create presentations, setting up the laboratory hardware, and making adaptations to the lpGBT emulator. The test results shown in Fig. 5.2 were also measured by him. Also, I want to thank Prof. Dr. Arnulf Quadt for enabling me to write the thesis at the II. Institute of Physics in Göttingen along with being the first referee, and Priv.Doz. Dr. Jörn Große-Knetter for helping me with organisational and remote access issues and being the second referee.

Erklärung nach §13(9) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:
Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.
Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den 28. Juli 2020

(Matthias Drescher)