CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering
Department of Control Engineering

Multichannel Signal Generator of Phase-Shifted Square Waves

Bachelor's thesis

Petr Brož

Supervisor
Ing. Martin Gurtner

2020

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Brož Petr**	Personal ID number:	**466185**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Multichannel Signal Generator of Phase-Shifted Square Waves

Bachelor’s thesis title in Czech:

Vícekanálový generátor fázově posunutých obdélníkových signálů

Guidelines:

Develop a 64-channel square wave generator with adjustable frequency common for all channels and with adjustable mutual phase shifts and duty cycles of individual signals. The frequency and phase shifts will be adjustable via a communication interface, and the refresh frequency of the settings must be at least 50 Hz. In addition, the generator must be capable of synchronization with other generators of the same type.

1. Research existing FPGA-based and ARM-based development kits the generator can be built upon. Choose the most suitable one.
2. Implement a standalone version (i.e., without the synchronization feature) of the generator on the chosen development kit.
3. Extend the standalone version by a feature allowing synchronization with other generators of the same type.
4. Assess the accuracy of the generated square waves.
5. Document the project on GitHub so that others can reproduce the generator.

Bibliography / sources:

[1] W. Beasley, B. Gatusch, D. Connolly-Taylor, C. Teng, A. Marzo, and J. Nunez-Yanez, “High-Performance Ultrasonic Levitation with FPGA-based Phased Arrays,” arXiv:1901.07317 [cs], Jan. 2019.
[2] A. Marzo, T. Corkett, and B. W. Drinkwater, “Ultraino: An Open Phased-Array System for Narrowband Airborne Ultrasound Transmission,” IEEE transactions on ultrasonics, ferroelectrics, and frequency control, vol. 65, no. 1, pp. 102–111, 2018.
[3] J. Zemánek, T. Michálek, and Z. Hurák, “Phase-shift feedback control for dielectrophoretic micromanipulation,” Lab on a Chip, vol. 18, no. 12, pp. 1793–1801, 2018.

Name and workplace of bachelor’s thesis supervisor:

Ing. Martin Gurtner, Department of Control Engineering, FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.12.2019** Deadline for bachelor thesis submission: ___________

Assignment valid until: **30.09.2021**

Ing. Martin Gurtner Supervisor’s signature	prof. Ing. Michael Šebek, DrSc. Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Petr Brož
March 2020

Acknowledgments

I would like to thank my supervisor Martin Gurtner for his detailed advice.

I thank Loi Do for letting me use his elegant LaTeX template.

I also thank my family for their support.

And last, but in no way least, I thank David Šibrava and Eliška Čukanová.
For everything.

Abstract

This bachelor's thesis describes the design and implementation of an open-source, modular, FPGA-based generator of square waves with configurable output frequency. The resulting device, built upon the widely available DE0-Nano kit, is suitable for applications in distributed manipulation. It provides 64 output channels, with the phase shift and duty cycle of each channel being adjustable with a resolution of 360 parts per period. These settings can be updated via UART at a frequency of 150 Hz. If only phase shifts or only duty cycles are being changed, the frequency rises to 300 Hz. In addition, the device is capable of synchronising itself with other devices of the same kind, which allows for more than 64 output channels to be used. To enable this, a custom PCB shield capable of interconnecting up to 4 generators has been designed. Finally, it is demonstrated that the output signals remain precise and synchronised even when generated by different devices in this way.

Keywords: multichannel signal generator, phase-shifted square wave, FPGA, VHDL

Abstrakt

Tato bakalářská práce popisuje návrh a implementaci modulárního open-source FPGA generátoru obdélníkových signálů s nastavitelnou výstupní frekvencí. Výsledné zařízení je založené na dostupném DE0-Nano kitu a je vhodné k využití v oblasti distribuované manipulace. Poskytuje 64 výstupních kanálů, jejichž fázové posuny a střídy jsou nastavitelné s rozlišením 360 dílků na periodu. Tato nastavení lze updatovat přes UART s frekvencí 150 Hz. V případě, že se mění jen fázové posuny nebo jen střídy, stoupá tato frekvence na 300 Hz. Zařízení je také schopné se synchronizovat s dalšími zařízeními stejného typu, což umožňuje použití více než 64 výstupních kanálů. K realizaci této synchronizace byl navržen propojovací PCB shield, který umožňuje propojit až 4 generátory. Na konci práce je ukázáno, že výstupní signály zůstávají přesné a synchronizované i v případě, že jsou takto generovány různými zařízeními.

Klíčová slova: vícekanálový generátor signálů, fázově posunutý obdélníkový signál, FPGA, VHDL

Contents

1 | Introduction

The motivation for this thesis came from applications in distributed manipulation, where continuous force fields are used to move objects without the need for physical contact. Those force fields are shaped by actuators distributed in space — and the greater the space to be influenced by them, the more actuators are needed. As the number of actuators grows into tens and hundreds, it becomes more and more challenging to generate the synchronised signals driving them. Thus, the need for a specialised signal generator arises. My goal was to satisfy this need by designing an open-source signal generator, capable of providing an arbitrarily large number of synchronised square wave signals suitable for use in such applications. The design is made available in github repository[1].

1.1 Primary use cases

In particular, two applications developed by my colleagues at CTU motivated the creation of this generator. Those are the acoustic manipulation platform AcouMan described in [10] and the dielectrophoretic micromanipulator described in [8]. The dielectrophoretic manipulator utilises 48 phase-shifted square waves at 300 kHz frequency to control its array of electrodes. AcouMan likewise uses phase-shifted square waves to drive its actuators. In its case, those were initially 64 ultrasonic transducers, arranged in an 8x8 grid and driven at a frequency of 40 kHz. During the development of my generator, the AcouMan platform has been extended to a 16x16 grid, requiring 256 such signals.

1.2 What is a phase-shifted square wave generator?

Allow me to define what I mean when I speak of the generator: A multichannel signal generator of phase-shifted square waves is a device with multiple digital outputs. Each output is a square wave with fixed amplitude, defined by three parameters: its frequency, its duty cycle[2] and its phase shift. The phase shift can be thought of as representing the delay of that signal's rising edge, measured against the rising edge of some reference square

[1]https://github.com/aa4cc/fpga-generator

[2]An observant reader might now remark that, since the duty cycle is a variable parameter, the outputs of the generator need not be square waves. They would be justified in doing so — strictly speaking, only signals with a 50% duty cycle are square waves. Perhaps it would then be better to call the device a generator of phase-shifted *pulse* waves. I nonetheless elect to call it a generator of square waves for two reasons — firstly, this is the name used by the assignment. Secondly, both primary use cases described above use the generator only to generate signals with a 50% duty cycle.

wave. All output signals have the same frequency. This shared frequency, as well as the duty cycles and phase shifts of all outputs, can be reconfigured on the fly by the user.

1.3 Goals and requirements

It goes without saying that the fundamental requirement is for the device to fulfil the definition above. Additional goals include, but are not limited to, those outlined in my assignment:

- **64 channels** — a single generator should provide this many phase-shifted square wave outputs, as mandated by the assignment.
- **Modular design (256 channels)** — to accommodate the plans for the extension of the AcouMan platform, an option must be provided to generate 256 synchronised outputs. This will be achieved by synchronising four generators, with each of them providing 64 channels.
- **Extendability beyond 256 channels** — although my implementation will only provide the synchronisation of 4 generators, the underlying design should be made easy to extend to any number of synchronised generators, allowing the number of output channels to be extended to any integer multiple of 64. In the ideal case, only the hardware used to connect the generators should need to be replaced, and few or none changes to the design of the device itself should be needed.
- **Synchronised output update (trigger)** — when the phase shifts and duty cycles of multiple synchronised generators are being reconfigured, the change in generated output signals should occur at the same time across all devices. Later in the text, I will refer to this event as *trigger*.
- **Configurable shared frequency** — the frequency of the outputs needs to be configurable by the user. To accommodate both use cases mentioned above, at least the frequencies of 40 kHz (for ultrasonic manipulation) and 300 kHz (for dielectrophoresis) must be supported. However, the device should support a wide range of output frequencies, so as to increase its usability for future applications.
- **Phase and duty cycle resolution of** 1° — a fine resolution of the phase shifts and duty cycles of the outputs is desirable, for it will allow to better approximate the continuous changing of those parameters. My goal was to achieve a resolution of 360 parts per period or 1°.
- **Refresh rate of 50 Hz** — this means that the phase shifts and duty cycles of output signals need to be reconfigurable with at least this frequency. The reason why a high refresh rate is needed is that the applications listed above make use of feedback control, which would be hindered by introducing delays in the form of high latency.

2 | Choosing the development kit

One of the first steps in designing the generator was choosing a suitable platform to build it upon. It was suggested to me to pick a development kit based either around an ARM or around an FPGA. In this chapter, I describe the reasons that led me to pick the DE0-Nano development kit.

2.1 Hardware requirements

There are a number of requirements that the development kit should fulfil in order to be suitable for the task at hand, among them:

- **Sufficient I/O count** - The generator needs 64 pins as outputs for its channels. In addition, two pins are needed for UART communication, and several more to facilitate the synchronisation of multiple generators. The exact number of pins used for synchronisation changed repeatedly during development, with the final version making use of 6 such pins.
- **Synchronous I/O toggling capability** - It is desirable that all the output pins are updated at the same time. If this does not happen synchronously, the accuracy of the generated phase shifts will suffer.
- **Parallel computation capability** - It is crucial that the logic dealing with the communication from the user runs in parallel to the functionality generating the phase-shifted signals. This is especially important in the case of multiple synchronised generators, as otherwise, communicating to one device would cause its signals to become desynchronised from those generated by other devices. This problem could be gotten around by the user always communicating to all synchronised devices at the same time; however, it would be preferable not to emburden the user with such a requirement.
  While not crucial, it is also desirable for the logic generating the individual 64 signals to run in parallel, since the signals are not dependant on one another.

2.2 Choosing between FPGA and ARM

2.2.1 Advantages of using ARM

Compared to FPGAs, ARM development kits are generally less expensive. Additionally, there is a far greater variety in the available kits (over a thousand search results at Farnell for ARM development kits compared to 55 results for FPGA development kits).

The ARM CPUs can be programmed using common sequential programming languages such as C, whereas FPGAs require the use of HDLs (hardware description languages), which are niche in comparison.

2.2.2 Advantages of using FPGA

FPGAs make it easy to parallelise an arbitrary amount of processes, unlike CPUs which are in this regard limited by the number of cores.

The synchronous update of all 64 outputs is also trivial to implement in an FPGA. I assume that achieving this on a 32-bit CPU would be more difficult in comparison. And indeed, the vast majority of ARM development boards are 32-bit. Those boasting 64-bit architectures tend to be expensive, negating one of the arguments in favour of ARM. Others, such as the Raspberry Pi, are 64-bit but do not offer a sufficient number of I/O pins.

Lastly, should the need for a CPU functionality arise, an FPGA can be used to satisfy it by adding a softcore microprocessor to the design. The inverse is not true.

2.2.3 Previous implementations of similar projects

The ultrasonic platform Ultraino described in [4] includes a signal generator with specifications similar to those of this project. It outputs 64 phase-shifted square waves at 40 kHz —this is the same frequency as in of the primary intended use cases of this generator (see 1.1). The Ultraino generator is built upon the Arduino Mega development kit. However, the results achieved using this processor-based setup fall short of those mandated by my assignment (those are listed in 1.3). The phase-shift resolution of Ultraino is 36° while controlling all 64 channels at 40kHz, with the update rate reaching 25Hz. Indeed, the authors themselves note that better phase-shift resolution and higher operating frequency could be reached by using an FPGA instead of a CPU.

An FPGA-based ultrasonic levitation platform described in [6] also contains a generator of phase-shifted square waves generating 40 kHz signals. The authors made use of a Xilinx Zynq SoC FPGA development kit. In a detailed report [7] [page 24] they describe their implementation of the signal generation by indexing a hardcoded 256-bit logic vector. It follows that the phase shift resolution of this generator is 256 divisions per period, or about 1.4°, which is significantly better than the results achieved by Ultraino.

2.2.4 Choosing FPGA

FPGA intuitively appears to be a suitable choice for this project, since FPGAs are typically used in applications that have high I/O counts, deal with fast signals and/or benefit from parallelisation. The generator, which is the subject of this thesis, has all 3 of those characteristics. Given the significant advantages of using an FPGA and the fact that a similar project using an FPGA achieved better results than one using a CPU, I have decided to use an FPGA as the platform upon which the generator will be built.

2.3 Choosing the DE0-Nano development kit

My reasons for choosing the DE0-Nano kit were the following:

- **Price** - The DE0-Nano sits at the cheaper end of the spectrum of available FPGA development kits. While even cheaper options do exist[1], they generally do not provide a sufficient I/O count, making DE0-Nano the least expensive kit that is still suitable for the job.
- **Suitable I/O organization** - The pins of DE0-Nano are split into two 40-pin headers on top of the board and a single 26 pin header on the bottom. Each of the top headers provides 36 usable I/O pins and the bottom header provides 16 such pins[2]. This setup lends itself nicely to the application—it will allow for the 64 output channels to be split into two sets of 32 outputs and routed to the top headers, with the bottom header left free to be used for synchronisation of multiple generators.
- **Previous experience** - I have previously worked with De2-115 and DE0 development boards. Those share not only the same manufacturer (being Altera products) but also the same FPGA family as the chosen DE0-Nano kit — all mentioned boards contain a Cyclone IV chip. Choosing this kit therefore allows me to continue using development software I am already accustomed to (Quartus and Modelsim).

Among the other boards considered were also the previously mentioned Xilinx Zynq SoC and the DE0-Nano SoC. The main advantage of those boards over the chosen kit is their inclusion of an ARM processor. The processor was utilised in [6], where the authors used it to calculate phase delays necessary to manipulate objects. However, no need for such complex calculations is expected to occur in my generator, making the CPU unnecessary. Both boards are also more expensive than the chosen kit.

[1]For example, the ICE40LP1K-BLINK-EVN, which can be obtained for half the price of a DE0-Nano.

[2]A significant portion of the bottom header is taken up by ADC input pins.

3 | Design overview

In this chapter, I aim to provide a big picture of the design. The chapter will consist of three parts. First, I will describe the structure of the device when operating in a standalone mode. An explanation of the mechanism by which the output signals are generated will follow. The final part will concern itself with the synchronisation of multiple generators.

The implementation details of individual parts of the design and of the synchronisation will be elaborated in further chapters.

3.1 Standalone generator

The design is split into several parts. A schematic outlining the operation of the device is shown in 3.1.

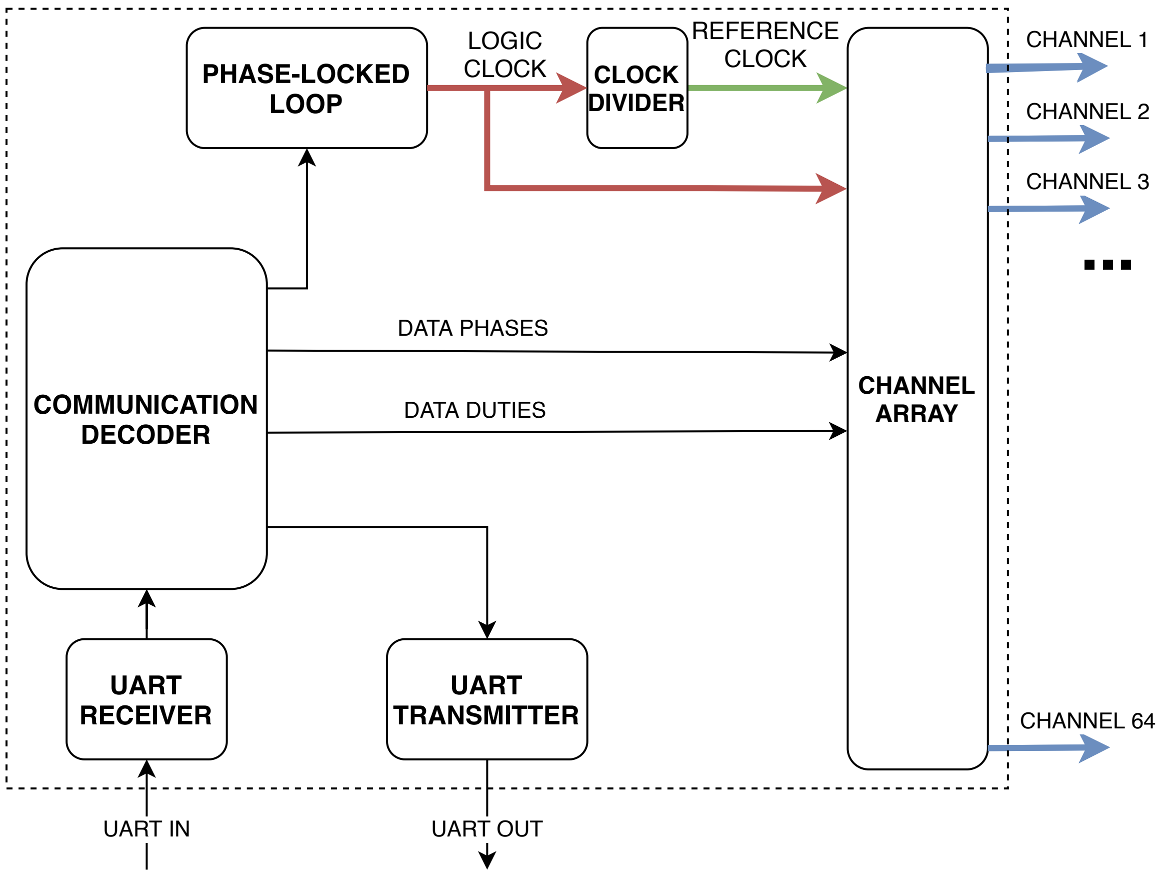


Figure 3.1: Simplified schematic of standalone generator

Note that the schematic is simplified and some parts of the design are omitted, as it is only meant to provide an overview of the basic function of the generator. A more detailed schematic will be provided in chapter 5.

Shown in blue are the output square waves. There are two noteworthy clocks in the schematic. The one shown in red will be referred to as the **logic clock** or the **fast clock**. This clock is generated by the phase-locked loop entity, and its frequency is equal to 360 times the output frequency of the generator. The other notable clock is the green **reference clock**, also called the **slow clock**. Its frequency is the same as the output frequency. This slow clock is used to synchronise the output signals — it can be thought of as the square wave against which the phase delays are measured. Those clocks, alongside data from the communication decoder, are used to generate the output signals within the channel array entity.

The device operates in the following fashion: first, data is sent from the user over uart. This data may represent one of several commands. For now, the relevant commands are frequency reconfiguration, phase shifts reconfiguration, and duty cycles reconfiguration.

Commands for reconfiguration of phase shifts or duty cycles contain within them the values of the parameter in question for all 64 channels. When such a command is received, the data is passed on to the channel array entity.

If the command calls for frequency reconfiguration, its contents are used to reconfigure the phase-locked loop entity. This changes the frequency of the logic clock, in turn also changing the reference clock and the frequency of the output signals.

After receiving the command from the user, the generator replies over uart, signifying which command the device received and whether it acted upon it or not.

3.2 Generating the phase-shifted signals

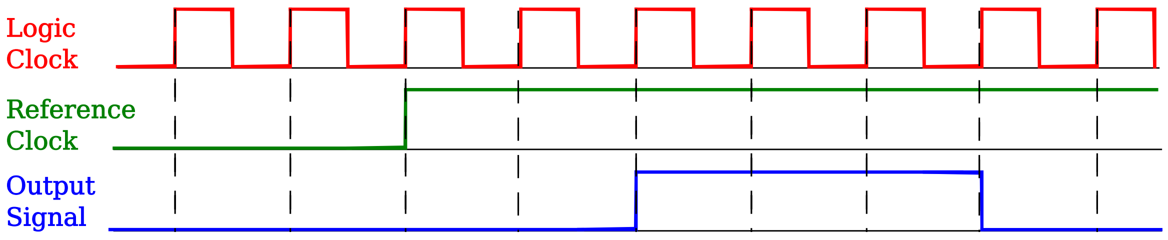


Figure 3.2: Signal generation for phase = 2°, duty = 3°

The generation of output signals makes use of the two clocks mentioned above — the logic and reference clocks. The frequency of the logic clock must be 360 times higher than that of the output signal in order to achieve the required resolution of 1°. Clearly, one period of the logic clock then corresponds to $\frac{1}{360}$ of the output signal's period. Thus, periods of the logic clock can be used to measure both the phase shift and the duty cycle of a generated output signal. This is the reason for calling the fast clock a logic clock — it is used in the design to clock the logic which generates the output signals. The slow clock (whose frequency is the same as that of the output signal) serves as a reference against which the phase shift is measured.

Apart from the logic clock and the reference clock, two integers are needed to generate the signal — one representing the phase shift and another representing the duty cycle, both in degrees. The values of those integers thus range from 0 to 360.

An example of signal generation is shown in figure 3.2. Here, the phase shift of the output signal is set to 2°, and its duty cycle is set to 3°.

3.3 Synchronising multiple generators

The purpose of using multiple generators at the same time is to increase the number of outputs beyond the 64 provided by a single device. The problem is that if multiple generators are used as they are, each will measure its phase delays against its own internal reference clock. This will cause the phases to float — a signal with zero phase shift generated in device A will be shifted in regards to a signal with zero phase generated in device B. It is then clearly necessary to synchronise the generators by means of some shared signal.

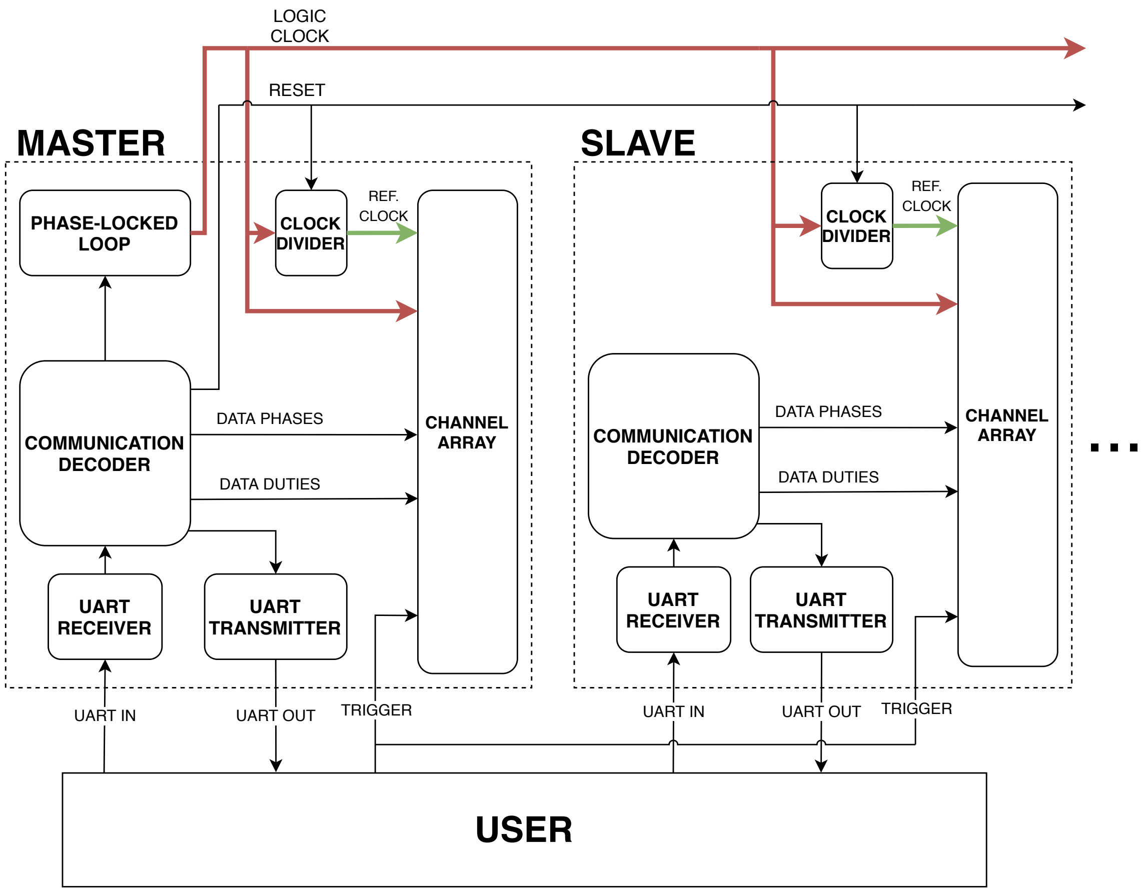


Figure 3.3: Simplified schematic of two synchronised generators

Additionally, it is not sufficient to only share the slow clock among the devices and to generate the fast clock internally. If this were done, the output signals of device A and device B would still be shifted. The shift would be caused by the difference in logic clock signals, which means that the outputs would at most be shifted by 1°.[1] This is nonetheless undesirable, so I choose to synchronise the devices by sharing the logic clock, as is shown in figure 3.3.

In this configuration, one device is designated master, while the other devices are slaves. Only the master device generates the logic clock, which is then shared with all slaves. Each device then derives the reference clock using a clock divider. A reset signal is introduced to allow for asynchronous resetting of all clock dividers. This is needed in case the dividers become desynchronised, which may happen if, for example, the logic clock connection is interrupted.

There is also a difference when it comes to reconfiguring the phases and duties of synchronised generators. When such data is sent to a standalone generator, it updates it's generated signals immediately. However, this is not possible to do when using synchronised generators. When the user sets out to update all synchronised generators with new phases and duties, they might not communicate to all of the devices at the same time — they might send the new data to the first generator, then to the second and so on. If the generators used the new data, their outputs would not update at the same time — a violation of one of the goals set out in the first chapter would occur.

To solve this issue, a trigger signal is introduced. This signal is driven by the user and shared among all devices. When multiple devices are synchronised, they only update their outputs when instructed to do so by the trigger signal. This allows the user to send data to all synchronised generators before causing an update by pulsing the trigger signal.

[1]It might seem that this problem could be solved by using a phase-locked loop as a frequency multiplier with a factor of 360. Then, only the slower reference clock could be shared, and each device would derive its own logic clock. This is sadly not an option, as the minimum input frequency of the phase-locked loop in Cyclone IV FPGAs is 5 MHz [1] [page 24].

4 | Communication protocol

This chapter will describe the protocol used to communicate with the device.

4.1 UART parameters and the trigger wire

The device communicates via two-way UART. Two pins are used: UART IN for communication from the user to the generator and UART OUT for replies sent by the generator.

The uart uses a standard configuration of one start bit, eight data bits and one stop bit. The baudrate used is 230400.

Additionally, when multiple generators operate in synchronized (chained) mode, a shared trigger wire is connected to all devices. This wire is driven by the user. Its signal is high while idle. When the user drives this trigger signal low, it signifies that the devices should update their generated signals with new parameters.

All I/Os of the generator (and therefore also the UART pins and the trigger wire) operate at 3,3V standard.

4.2 User to device communication

Each command sent by the user to the device consists of following three parts:

- **CODE** — 1 byte, used to identify the command
- **DATA** — 0, 18 or 72 bytes, depending on command type
- **CRC** — 1 byte, generated as crc8 with generating polynomial 0x07 from CODE and DATA.

A list of all possible commands follows.

4.2.1 Command: Set phases

The **CODE** of this command is **0x01**.

This command is used to configure the phase shifts of all 64 square waves generated by the device. The phase of each channel is represented by a 9-bit unsigned integer ranging from 0 to 360.

The **DATA** of this command is 72 bytes long. It consists of concatenated 9-bit integers in ascending order of the channels — so that the first 9 bits of the data represent the phase shift of the first channel, the second 9 bits that of the second channel and so on.

Note that, unless the device is operating in standalone mode, the generated square waves are not updated with the configuration sent by this command until the user pulses the external trigger signal.

4.2.2 Command: Set duties

The **CODE** of this command is **0x02**.

This command is used to configure the duty cycles of all 64 square waves generated by the device.

It's **DATA** has the same format as that of the set phases command.

Note that, unless the device is operating in standalone mode, the generated square waves are not updated with the configuration sent by this command until the user pulses the external trigger signal.

4.2.3 Command: PLL reconfig

The **CODE** of this command is **0x04**.

This command is used to reconfigure the phase-locked loop, which is used within the master device to generate the fast clock.

The **DATA** of this command consists of an 18-byte long reconfiguration scanchain. The format of this scanchain is described in [2] [page 99].

4.2.4 Command: Inquire master

The **CODE** of this command is **0x08**.

This command is used to request a reply that will tell the user whether the device in question is master or slave.

It has no **DATA**.

4.2.5 Command: Synchronise dividers

The **CODE** of this command is **0x10**.

This command is sent to the master device to request the synchronization of clock dividers of all devices.

It has no **DATA**.

4.3 Device to user communication

The device only sends messages to the user as replies to their commands. Each such reply is exactly one byte long and consists of two parts:

4.3.1 Low nibble

The low nibble identifies which command the device is replying to. For some commands, it may also carry additional information. The possible values and their meanings are:

- **0x1** — reply to set phases command.
- **0x2** — reply to set duties command.
- **0x3** — reply to pll recofig command.
- **0x4** — reply inquire master command. Signifies that the device is master.
- **0x5** — reply inquire master command. Signifies that the device is slave.
- **0x6** — reply to synchronise dividers command.
- **0x7** — reply to synchronise dividers command. Signifies that the command was ignored because the device is not master.
- **0x8** — signifies that an invalid command code was received.

4.3.2 High nibble

The high nibble of the reply is used to signify whether the CRC sent by the user was found to be correct or not. A value of **0xF** means that the CRC was correct. In case the CRC was not right, **0x0** is sent instead.

As a rule, the generator does not act upon commands with invalid CRC. An exception to this is the inquire master command — a reply to this command is sent in the same fashion even if its CRC happens to be incorrect.

Another exception is the unknown code reply. If the device does not recognize the command code, it does not calculate CRC — so, in this case, the high nibble is meaningless and should be disregarded.

4.4 Refresh rate

One of the requirements set out by the assignment is that a refresh rate of 50 Hz be reached when reconfiguring the phase shifts and duty cycles. To reconfigure either the phases or the duties of a device, the user sends 74 bytes and receives one byte as a reply, totalling 75 bytes. Given that the UART uses one start bit and one stop bit for each byte sent and that the BAUD rate is 230400, the total time it will take to send 75 bytes is calculated as:

$$t = \frac{10 \cdot 75}{230400} \approx 3.26 \text{ ms} \tag{4.1}$$

This corresponds to a refresh rate of about 307 Hz if only one set of parameters is being changed, or about 153 Hz if both phases and duties are being reconfigured. Here I assume that the time it takes to transmit the data will be the dominant factor — that any delays caused by the device can be neglected. I think this assumption is justified, because, as will become clear in the next chapter, the device does not perform any complicated calculations on the data. The most complex task done by the device is calculating the CRC from received data; however, as will be described in section 5.3.11, this is done in parallel while the CRC byte is being received.

5 | Implementation

In this chapter, I provide a detailed description of the generator implementation, including details on individual design parts.

A schematic of the implementation is shown in figure 5.1. This schematic is significantly more complete than the previous simplified one (provided in figure 3.1). However, I still choose to omit some internal connections in hopes of making the schematic more readable. For a complete list of inputs and outputs of individual blocks, I refer the reader to section 5.3.

Drawn in red and green are the logic clock and reference clock signals discussed previously. Yellow is used to highlight input and output pins of the device, with rectangular yellow shapes being reserved for LEDs and DIP switches mounted upon the DE0-nano board. Non-rectangular yellow shapes correspond to GPIO pins.

With the exception of section 5.4, whose nature justifies it, I refrain from discussing pin mappings of individual inputs and outputs in this chapter. A complete list of pin mappings used in the design is instead provided in appendix A. The labels used within the appendix correspond to those used in figures throughout this chapter.

Each block within the schematic is implemented as a VHDL entity, or as an instance of an available Altera megafunction, or as a combination of those. Individual blocks will be elaborated in section 5.3.

My implementation allows for up to four generators to operate at the same time, as this was the number required by my colleagues. Should the need for more than four synchronised generators arise in the future, only two parts of the implementation will need to be changed, one of them being the synchronisation PCB shield described in section 5.4. Possibly, another slight change may be necessary in the Communication decoder entity, as will be discussed in section 5.3.12. The rest of the design makes no assumptions as to the number of devices operating in a synchronised fashion.

5.1 Operating modes

The device may operate in one of three modes: **standalone**, chained (=synchronised) **master** and chained **slave**.

When only one generator is used, it operates in **standalone** mode. This is the case when only the 64 outputs provided by a single generator are needed.

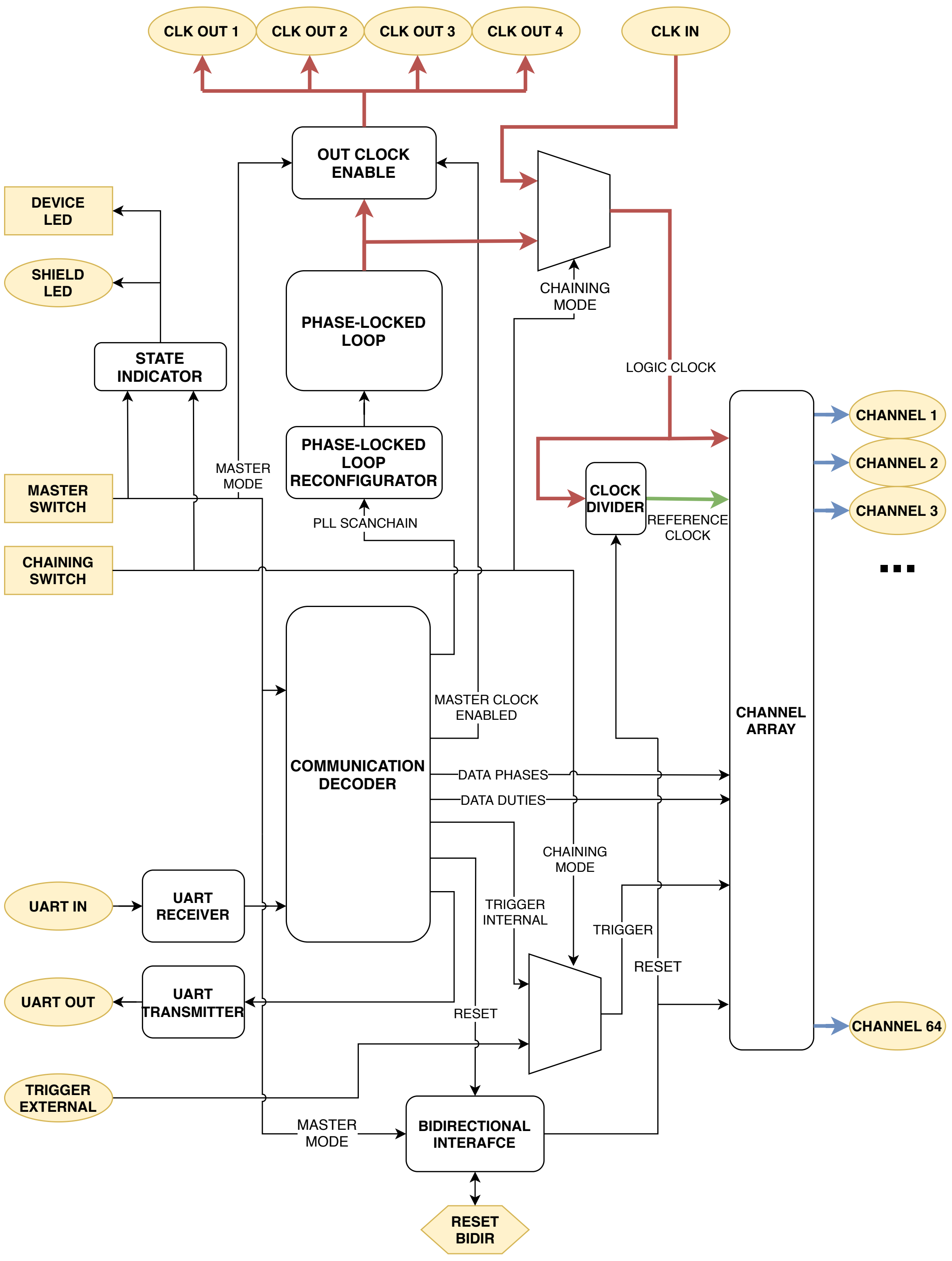


Figure 5.1: Detailed schematic of the generator

If more than 64 channels are desired, multiple generators operate together — they are chained. In this case, the device can either be operating as the **master** or as one of the **slave**s.

My implementation can fulfil all 3 of those roles. Multiple solutions for switching between the modes were used during development. The final version uses two DIP switches to select the mode — those are marked master switch and chaining switch in figure 5.1. When the chaining switch is in the position corresponding to standalone mode, the position of the master switch does not matter, since the logic driven by its signal (labelled master mode in the schematic) only influences parts of the design that are irrelevant when the device operates alone.

The current mode of the device is signalised to the user via two LEDs — one on the DE0-nano board, and another on the PCB synchronisation shield, which will be discussed in section 5.4. The modes are signalized by:

- **LED on** — meaning that the device is in **slave** mode.
- **LED off** — meaning that the device is in **master** mode.
- **LED blinking** — meaning that the device is operating in **standalone** mode.

5.1.1 Chaining switch function

As shown in the schematic, the signal driven by the chaining switch is connected to two multiplexers. The upper multiplexer deals with the logic clock — if the generator is in one of the synchronised modes (chaining switch is on), logic clock is taken from the CLK IN input pin. Note that this external clock is used even if the device in question is master. While in standalone mode, the logic clock signal generated by the internal phase-locked loop is used instead.

The lower multiplexer determines which trigger signal is passed to the channel array. When operating in one of the two synchronised modes, external trigger signal from the user is passed; otherwise, the internal trigger signal generated by the communication decoder is used.

5.1.2 Master switch function

The master switch serves to determine whether the device transmits or only receives from the bidirectional pin (labelled reset bidir in the schematic). It also drives the out clock enable block — the generated logic clock signal can only be passed to the CLK OUT pins if the device is in master mode. Lastly, it drives the communication decoder entity, which modifies its reactions to some commands depending on whether it is operating as master or as slave.

5.2 Connecting multiple generators

Figure 5.2 shows the way the generators are connected together when operating in chained mode[1]. The schematic is complete with the exception of ground connections, which are not shown.

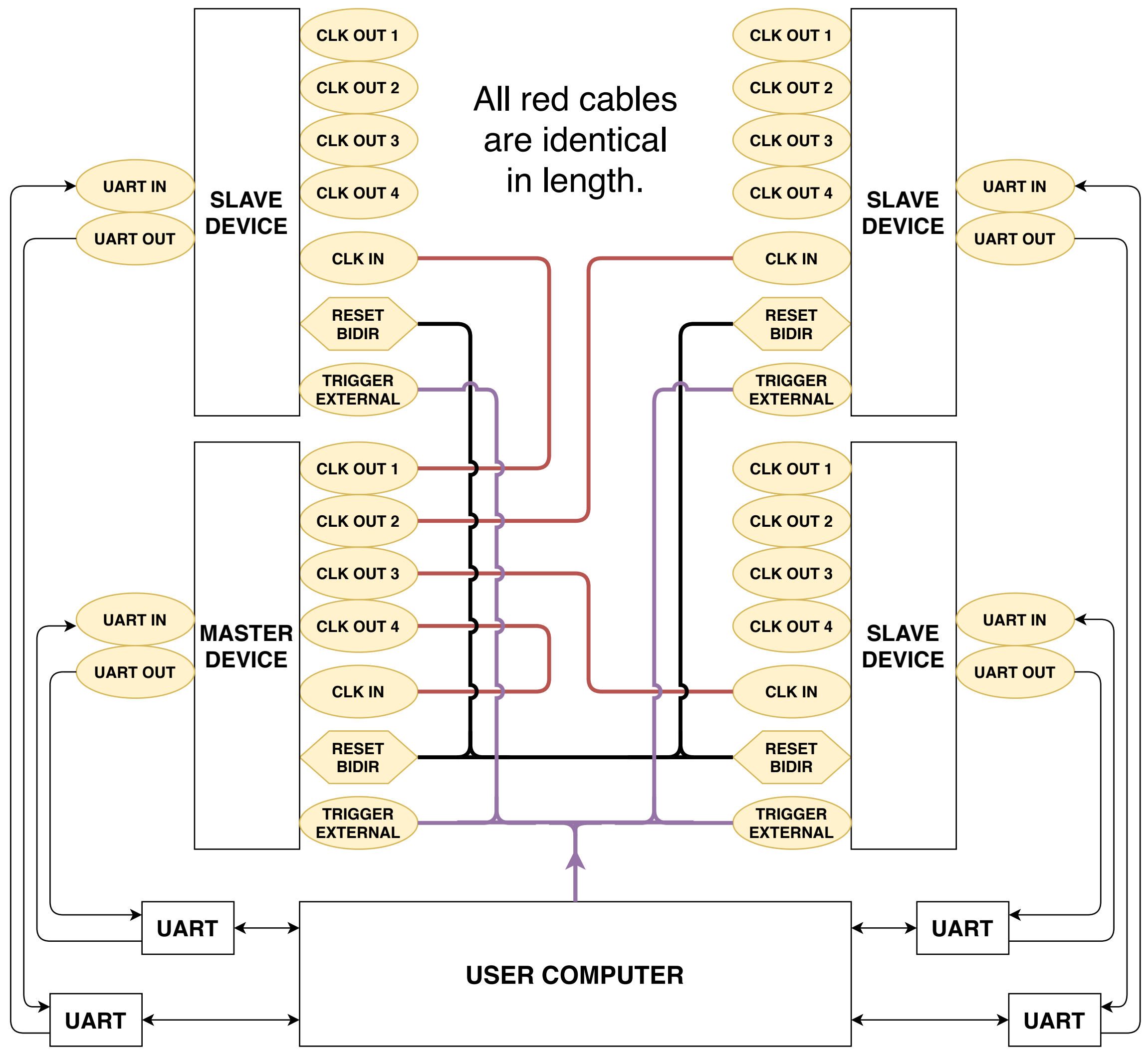


Figure 5.2: Schematic of connections between 4 synchronised generators

The bidirectional reset pins of all devices are connected to a shared wire. For details on how the bidirectional pins are used, see description of bidirectional interface entity in subsection 5.3.4. Additionally, the trigger external pins of all devices are connected to another shared wire driven by the user (shown in violet in the schematic).

[1]Note that the device pins facing inward in the schematic, that is all shown pins apart from UART ones, are connected to components on the custom PCB shield, which will be discussed in section 5.4.

The red connections are realised using identical coaxial cables. This is done to ensure equal transmission delays for all devices. Unequal transmission delays would lead to the logic clock signals (which are transmitted along those cables) not being synchronised across the devices. Thus, the outputs of individual devices would also desynchronise.

This is also the reason why the logic clock signal is not routed internally in the master device when operating in chained mode — doing so would desynchronise the master from its slaves.

Finally, each device communicates to the user via its own UART. In a previous iteration of the design, another option was considered where only the master device communicated. It would receive settings for itself as well as for all of its slaves, passing this information to them. However, the current implementation is faster (communication can be parallelised) and simpler (the master device needs not to know how many slaves are connected to it).

5.3 VHDL implementation

The generator is implemented as a Quartus II project. The top design entity is a schematic file named *ShiftedSignalGenerator.bdf*. This file is used to connect individual entities — those entities, for the most part, correspond with the blocks shown in figure 5.1. What follows is a description of the entities.

5.3.1 Channel array

Implemented in file *ChannelArray64.vhd*.

```
ENTITY ChannelArray64 IS
PORT
(
CLK_LOGIC : in std_logic;
CLK_FREQ : in std_logic;
DATA_PHASE_IN : in std_logic_vector(575 downto 0);
DATA_DUTY_IN : in std_logic_vector(575 downto 0);
TRIGGER : in std_logic;
RESET : in std_logic;
OUT_SIGNAL : out std_logic_vector(63 downto 0)
);
END ChannelArray64;
```

This entity holds within it 64 instances of Channel entities, with each instance being responsible for generating one of the output signals. Its first two inputs are the aforementioned logic and reference clocks — those, along with the RESET signal, are provided to all channel instances. The DATA_PHASE_IN and DATA_INPUTS_IN vectors each contain 64 9-bit integers. The entity splits the vectors into individual integers, which are passed to the channel instances upon the rising edge of the TRIGGER signal.

5.3.2 Channel

Implemented in file *Channel.vhd*.

```
ENTITY Channel IS
PORT
(
CLK_LOGIC : in std_logic;
CLK_REF : in std_logic;
PHASE_SHIFT_IN : in std_logic_vector(8 downto 0);
PULSE_WIDTH_IN : in std_logic_vector(8 downto 0);
RESET : in std_logic;
OUT_SIGNAL : out std_logic
);
END Channel;
```

This entity implements the square wave generation described in section 3.2. It makes use of two processes — one for measuring the phase shift and another for measuring the duty cycle.

5.3.3 Clock divider

Implemented in *clock_divider.vhd*.

```
ENTITY clock_divider IS
GENERIC (clock_divisor_factor: integer := 360);
PORT (
clk_in : in std_logic;
reset : in std_logic;
clk_out: out std_logic
);
END clock_divider;
```

This is a simple entity implementing clock division, consisting of little more than a counter with asynchronous reset.

5.3.4 Bidirectional interface

Implemented in *bidir_interface1.vhd.*

```
ENTITY bidir_interface1 IS
PORT (
 bidir_pin : inout std_logic;
 sig_use : out std_logic;
 sig_in : in std_logic;
 is_master : in std_logic
 );
END bidir_interface1;
```

This entity controls the bidirectional reset pin. As shown in figure 5.2, those pins are connected to a wire shared among all devices.

The entity always outputs the value read from bidirectional reset pin to sig_use. The sig_use output drives the clock divider and channel array entities [2]. When the device is not operating in master mode (value of the is_master input is 0), the pin is kept in high impedance. Devices operating in master mode drive the pin (and thus the shared wire) with the value read from sig_in (the sig_in input is driven by communication decoder).

A problem could arise if multiple master devices were connected to the shared wire — one might drive the shared wire high while another drove it low, damaging the devices. To prevent this, internal pullup resistors are enabled on the bidirectional pin. The master device only ever drives the pin low or sets it to high impedance.

As a last detail, the signal is inverted before and after the bidirectional interface entity. Inside the device, the reset signal is treated as active-high, but it is passed over the shared wire as active-low.

[2]See figure 5.1, signal labeled RESET

5.3.5 Uart receiver

Implemented in *uartRec.vhd*.

```
ENTITY uart_rx IS
GENERIC (
DATA_BITS : integer := 8;
UART_BAUD_RATE : integer := 230400;
TARGET_MCLK : integer := 50000000);
PORT (
 clock : in std_logic;
 data : out std_logic_vector(DATA_BITS-1 downto 0);
 data_valid : out std_logic;
 rxd : in std_logic
 );
END uart_rx;
```

This entity implements a uart receiver. It is rather unremarkable — after receiving the data byte, it passes it on to the communication decoder entity and drives the data_valid signal high. It communicates with 8 data bits, one stop bit and no parity bits. A different baud rate can be chosen by editing its generics. The baud rate used in my implementation — 230400 — is by no means the maximum usable value. I have successfully communicated to the device at baud rate 460800. However, my colleagues encountered problems when they tried to communicate at this baudrate from a Raspberry Pi. I have thus reverted to 230400.

5.3.6 Uart transmitter

Implemented in *uartTr.vhd*.

```
ENTITY uart_tx IS
GENERIC (
DATA_BITS : integer := 8;
STOP_BITS : integer := 1;
UART_BAUD_RATE : integer := 230400;
TARGET_MCLK : integer := 50000000);
PORT (
 clock : in std_logic;
 start : in std_logic;
 data : in std_logic_vector(DATA_BITS-1 downto 0);
 trx : out std_logic;
 ready : out std_logic
 );
END uart_tx;
```

This is an implementation of a standard uart transmitter. It's parameters are the same as those of the previously discussed receiver.

Upon the rising edge of the start signal, it transmits the contents of data over uart. The ready signal is used to inform the communication decoder entity that the transmitter has finished transmitting. Since only one-byte messages are ever transmitted by the device (see section 4.3), I saw no need to implement an input data buffer.

5.3.7 State indicator

This block is not implemented as an entity but rather as a multiplexer and clock divider within the top design schematic. A clock divider creates a slower clock from the onboard 50 MHz clock source and the chaining mode signal is used to multiplex between two modes — if chaining is off, the divided clock is passed to the indication LEDs. Otherwise, the master switch signal is passed to the diodes.

5.3.8 Phase-locked loop

This entity was not implemented by me. It is an instance of the Altera ALTPLL megafunction [3]. The wizard-generated file corresponding to this entity is pll180.vhd.

```
ENTITY pll180 IS
PORT (
 areset : IN STD_LOGIC := '0';
 configupdate : IN STD_LOGIC := '0';
 inclk0 : IN STD_LOGIC := '0';
 scanclk : IN STD_LOGIC := '1';
 scanclkena : IN STD_LOGIC := '0';
 scandata : IN STD_LOGIC := '0';
 c0 : OUT STD_LOGIC ;
 locked : OUT STD_LOGIC ;
 scandataout : OUT STD_LOGIC ;
 scandone : OUT STD_LOGIC
 );
END pll180;
```

This entity is used to generate the logic clock. The inclk0 input is driven by the 50 MHz clock source, with all remaining inputs being driven by the phase-locked loop reconfigurator entity. Of the outputs provided by the entity, only the c0 one is used — this being the logic clock output.

5.3.9 Phase-locked loop reconfigurator

Implemented in *pllReconfigurator.vhd*.

```
ENTITY PllReconfigurator IS
PORT (
 clock : in std_logic;
 performReconfig : in std_logic;
 scanchain : in std_logic_vector(143 downto 0);
 areset : out std_logic;
 scanclk : out std_logic;
 scandata : out std_logic;
 scanclkena : out std_logic;
 configupdate : out std_logic
 );
END PllReconfigurator;
```

This entity implements a simple state machine used to reconfigure the phase-locked loop. The scanchain and performReconfig inputs are driven by the communication decoder entity. All outputs are used to drive the inputs of the phase-locked loop.

Upon the rising edge of the performReconfig signal, contents of the scanchain input are read by the entity and then shifted serially into the phase-locked loop.

5.3.10 Out clock enable

Implemented in *outClockEnable.vhd*.

```
ENTITY outClockEnable IS
PORT (
 clock_in : in std_logic;
 is_master : in std_logic;
 out_enable : in std_logic;
 clock_out : out std_logic
 );
END outClockEnable;
```

This entity is basically a 3-input AND gate. When both is_master (driven by the master switch) and out_enable (driven by the communication decoder) are high, clock_in is passed on to clock_out. The main purpose of this entity is to allow the master generator to temporarily turn off its clock outputs, which is used when executing a command to synchronise dividers.

5.3.11 Communication decoder

Implemented in *CommV2.vhd.*

```
ENTITY CommV2 IS
PORT
(
CLK : in std_logic;
byte_in : in std_logic_vector(7 downto 0);
byte_in_valid : in std_logic;
uart_ready : in std_logic;
uart_send_byte : out std_logic_vector(7 downto 0);
uart_send_start : out std_logic;
chan_phases : out std_logic_vector(575 downto 0);
chan_duties : out std_logic_vector(575 downto 0);
master_clock_enable : out std_logic;
master_in : in std_logic;
pll_scanchain : out std_logic_vector(143 downto 0);
pll_perform_reconfig : out std_logic;
reset : out std_logic;
trigger_internal : out std_logic
);
END CommV2;
```

This entity implements the communication protocol described in chapter 4.

Inputs and outputs

The entity is clocked by the onboard 50 MHz clock. The byte_in and byte_in_valid inputs are driven by the uart receiver entity. The uart_send_byte and uart_send_start outputs are used to pass data to the uart transmitter entity, which in turn drives the uart_ready input.

Two 72-byte vectors — chan_phases and chan_duties — are used to pass the phase shift and duty cycles settings to the channel array entity. The trigger_internal signal is used to update the channels with those settings — but only when the device is operating in standalone mode (when this is not the case, an external trigger signal is used, as shown in figure 5.1).

A shorter, 18-byte vector named pll_scanchain is used to pass the pll reconfiguration scanchain to the phase-locked loop reconfigurator entity. Along with it, the pll_perform_reconfig output is used to inform the reconfigurator entity that it should shift this data into the phase-locked loop itself.

The master_in input is driven by the master switch and is used to inform the decoder whether it is in master mode or not.

The master_clock_enable output drives the out clock enable entity. It allows the decoder to turn off the sharing of clocks via CLK OUT pins. This is used in combination with the reset output to perform the synchronisation of clock dividers among all chained devices.

Structure

The entity consists of two finite state machines; both implemented as VHDL processes running in parallel. One of them, called the decode process, is diagramed in figure 5.3. The decode process is responsible for evaluating incoming communication, acting upon it and assembling replies to be sent back to the user.

The second, far simpler process, is named calculateCrc. As its name would suggest, it is used to calculate a CRC — at one point (see gray rectangle in figure 5.3) execution of the decode process, data is passed to the calculateCrc process. Upon completion of the calculation, the result is passed back to the decode process.

The decode process

The process begins in the oval in the upper left corner of figure 5.3 labelled decoder ready. A byte is received — this is assumed to be the code byte. If it corresponds to one of the three commands carrying data, a corresponding number of bytes is received and saved in the entity's data buffer.

We then arrive at the request CRC calculation step — the code byte, as well as the contents of the data buffer (if any), are passed to the calculateCrc process. While the calculateCrc process is working, the last byte — that being the CRC which terminates each command from the user — is received. Once the user-supplied CRC has been received and the calculated CRC has been returned by the parallel process, they are compared. The process then updates its high nibble variable accordingly.

Having set the variable, it is now time to act upon the command. The easiest case is if the command in question is inquire master (code 0x08). The device simply consults the master_in input and sets the low nibble variable.

In all 3 cases of commands carrying data, the procedure is similar — that is why within the diagram, I have grouped them into a single path. The trigger_internal signal is pulsed if the command is either set phases (code 0x01) or set duties (code 0x02). In case of a PLL reconfig command (code 0x04), pll_perform_reconfig is pulsed instead. Note that, should the user-supplied CRC not match that calculated previously, no action is taken.

The last case — the 0x10 code — will be discussed later.

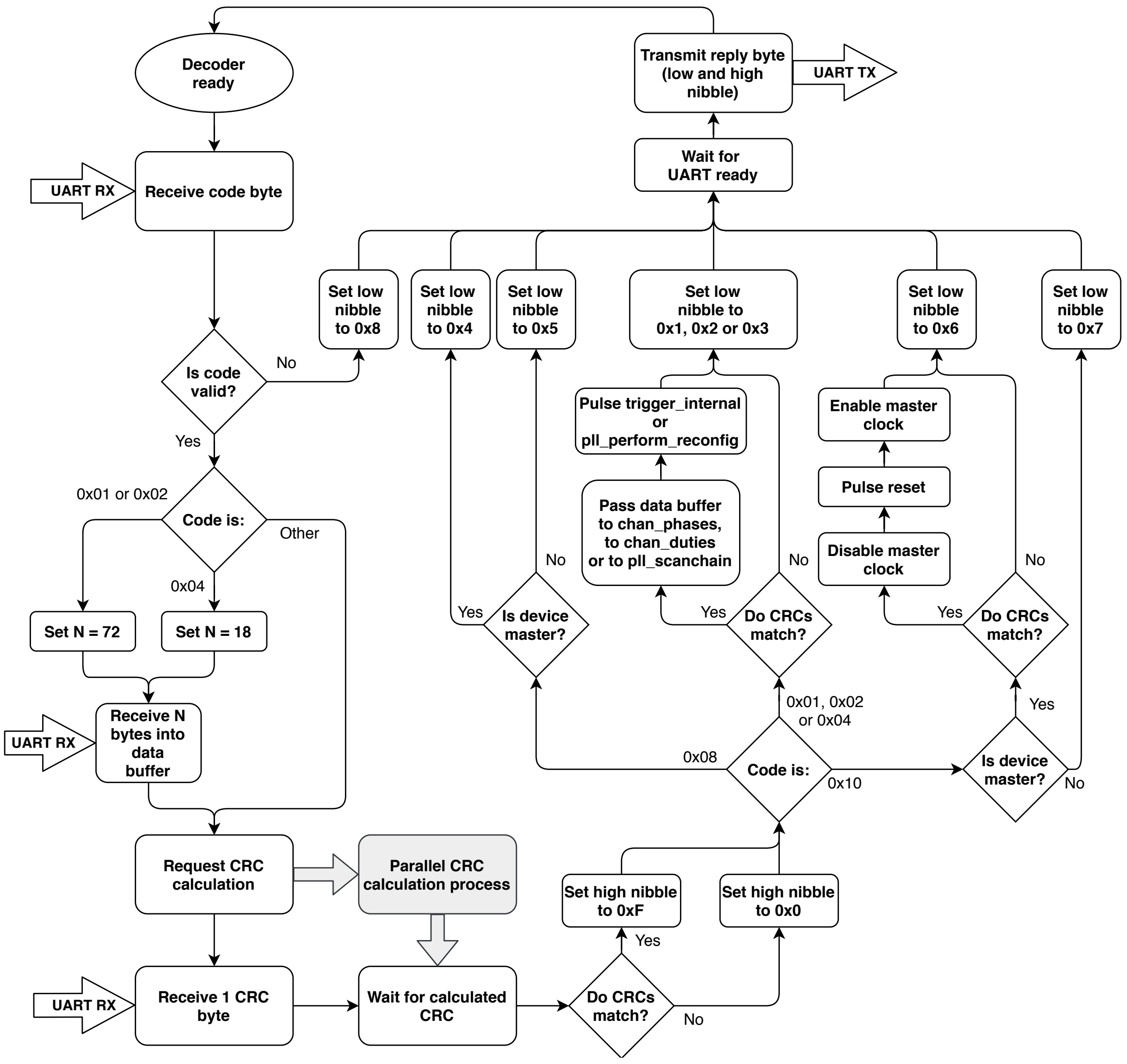


Figure 5.3: Decode process diagram of communication decoder entity

Upon executing the command (whatever it happens to be), the process proceeds to set the low nibble variable to its appropriate value. Then it waits for the uart_ready signal to be high (in case the uart transmitter was busy), after which it assembles a reply byte from the low nibble and high nibble variables and passes it to the uart transmitter entity. Then, it is ready to receive a new command.

Worth noting is the fact that if a byte received as a code is not recognised, the process goes directly to set low nibble to 0x8 and to transmit a reply. As noted previously in the description of the communication protocol, the high nibble should be disregarded in this case — its value will be left over from the previous command. Indeed, no CRC is received (nor calculated) in this case.

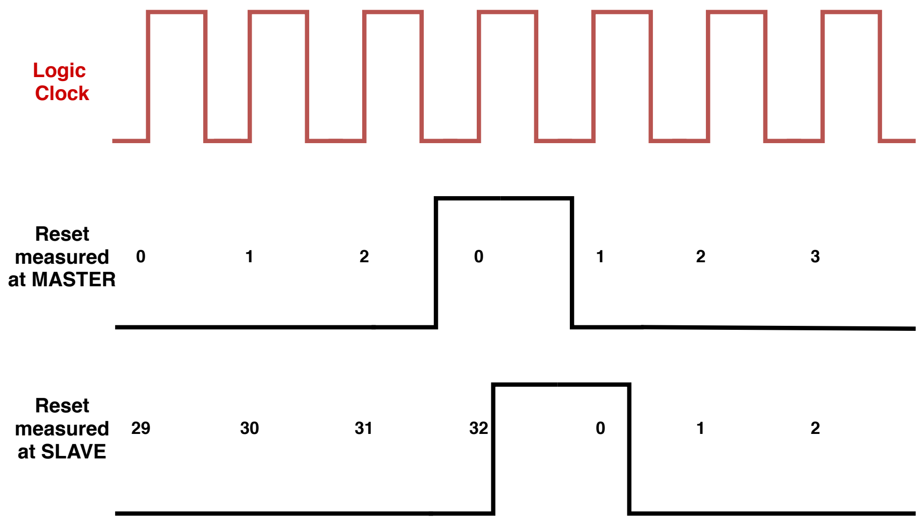


Figure 5.4: Failed divider synchronisation without disabling master clock output

The code 0x10 corresponds to a synchronise dividers command. The goal of this command is to reset clock dividers (see figure 5.1) and thus eliminate their possible desynchronisation. Those clock dividers are driven by the logic clock signal (shown in red). However, before sending the reset command, it is first necessary to disable the master clock outputs. Figure 5.2 shows the connections between the generators — note that while the logic clock signal path from master to each slave (and from master to master itself) is the same, signal paths for the reset signal differ. This means that some devices will receive the reset signal later. If the logic clock were enabled during this time, the delay might cause the synchronisation to fail.

Figure 5.4 illustrates such a failiure. The shown integers represent the counter values within the clock divider entity. At the start, the counters are desynchronised. The counter increments with each rising edge of logic clock [3]. When reset arrives in the device, this counter is set to zero — however, the different delay of the reset signal may cause a rising edge of logic clock to increment a counter in one generator but not in another. So, after resetting, the counters remain unsynchronised.

A solution is shown in figure 5.5. By disabling the logic clock while counters are reset, the failure is averted.

5.3.12 Extending the design and the Communication Decoder entity

As was described in the preceding section, one of the commands results in the clock dividers being synchronised across all devices. To achieve this, the master clock is disabled for some

[3]Note that the logic clock signal is the same whether we measure it at master or slave since its signal paths are the same.

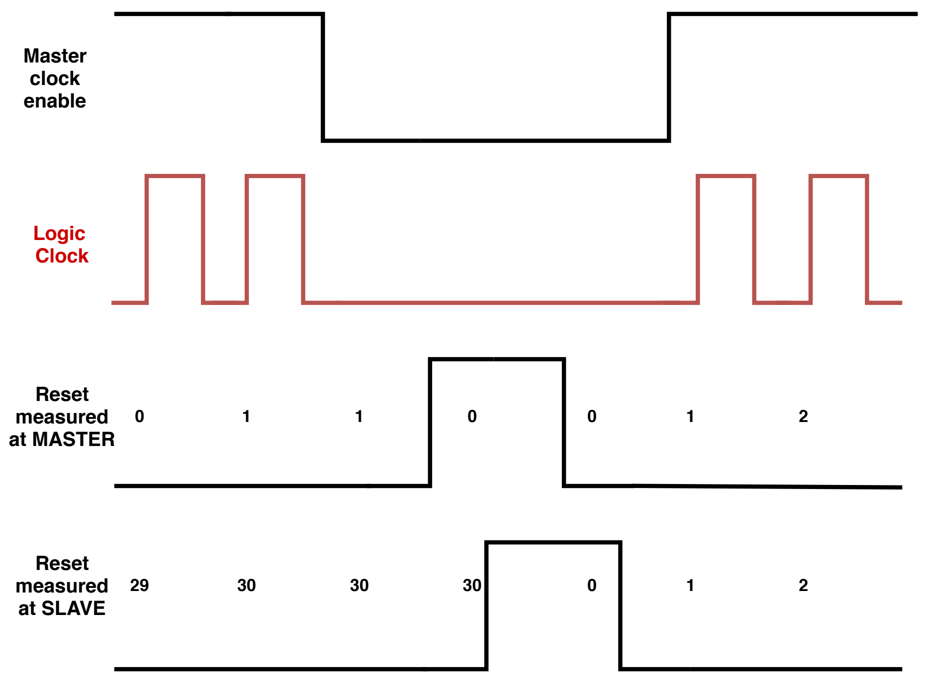


Figure 5.5: Succesful divider synchronisation with disabled master clock output

time. This time period needs to be sufficiently long so as to ensure that, by the time the master clock is enabled again, the reset signal has already reached all slaves.

The duration for which the master clock is disabled is governed by a constant within the *commV2.vhd* file named enableCount . It is an integer representing the number of clock cycles of the onboard 50 MHz clock for which to keep the clock output disabled. In my implementation, I have set its value to 50000, which corresponds to a clock disable duration of one millisecond.

This is more than enough in my implementation with 4 synchronised generators. However, if the number of synchronised generators were to be drastically increased, it is in theory possible for the transmission line length to become so long that one millisecond is no longer sufficient. Therefore, should this design ever be adapted to use an extremely large number of synchronised devices, the enableCount constant might need to be increased.

5.4 Synchronisation shield PCB

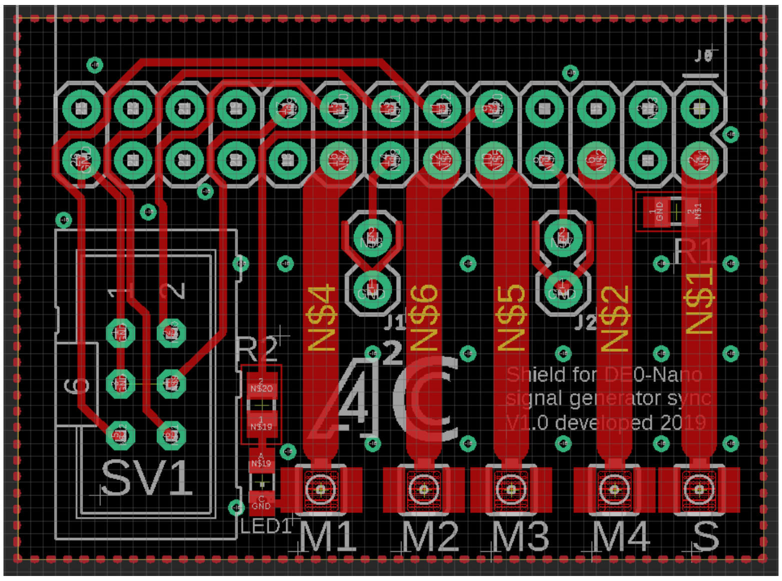


Figure 5.6: Screenshot of synchronisation shield board layout (top side) taken in Autodesk Eagle

To facilitate the connections described in section 5.2, I designed a custom PCB shield, the layout of which is shown in figure 5.6. Its design files are made available in the github repository[4], in folder chaining_shield. The dimensions of the PCB are 36.81 by 26.72 millimetres. The shield mates with DE0-Nano's bottom 2x13 pin header. When multiple (up to 4) devices operate together, each DE0-Nano board will be equipped with one of those shields. The shields themselves are then connected by means of one 6-wire flat cable and up to 4 coaxial cables (the coaxial cables are shown in red in figure 5.2). For determining the orientation of the shield in regards to the GPIO header, observe the leftmost bottom pin in figure 5.6 — this pin mates to the ground pin of DE0-Nano's header. In the space not occupied by components or traces, a ground plane is spilt. The board is double-sided, with its bottom side consisting entirely of a ground plane, connected to the top ground plane by use of vias. A description of the components follows, going in order from left to right.

5.4.1 Flat cable connector

The flat cable connector is marked SV1 in the figure. Although only three wires are required by the design, a 6-wire flat cable was used instead due to component availability. The top left pin, and the one underneath it, are used to share ground among the devices. [5] The top

[4]https://github.com/aa4cc/fpga-generator

[5]I used two ground wires in order to surround the third wire in hopes of shielding it from interference. In a previous iteration of the design, this surrounded wire (now used for the reset signal) carried the reference clock.

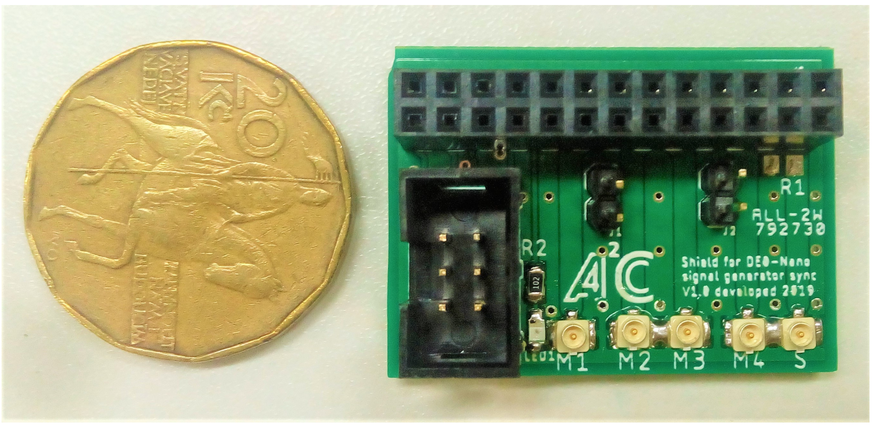


Figure 5.7: Populated synchronisation shield with a 20 CZK coin for scale

right pin connects to pin F14 of the DE0-Nano board — the bidirectional reset pin. The right pin of the middle row connects to pin G15, which is used as the external trigger input — the input driven by the user.

The bottom row of the flat cable connector connects to pins F15 and D15, which are not used in the design.

5.4.2 Indicator LED

The indicator LED is driven by pin C15. It is used to indicate whether the device is operating as master, slave or as a standalone device.

5.4.3 Coaxial cable connectors

The coaxial cable connectors are labelled M1 to M4 and S. The M connectors are the master clock outputs — labelled CLK OUT in figures 5.1 and 5.2. The connector labelled S is the clock input — labelled CLK IN in aforementioned figures.

The clock outputs connect to pins B16, D16, D14 and G16 if the DE0-Nano board. The decision to use those particular pins was not arbitrary — I chose them so as to allow all traces connecting the clock output pins to their respective coaxial connectors to have the exact same shape. This is done to keep the transmission line lengths the same. Due to the dimensions of the coaxial cable connectors, this could not have been achieved had I chosen four neighbouring pins. The input connector links to pin E17.

Additionally, a resistor can be observed connecting the S connector's trace to the ground plane near the E17 pin. I added this to the board in case a termination resistor was needed for impedance matching. However, I left this resistor footprint unpopulated in the end. The reason for doing so was twofold — firstly, when used at the output frequency of 40 kHz (which was the primary intended use case), the absence of the resistor did not appear to hinder the device's function in any way. Secondly, populating the footprint with a resistor resulted in severe deterioration of the fast clock signal — to the point that the device ceased to function properly. Since a generic resistor was used, I suspect that its parasitic elements might have been the cause. However, I did not investigate this further.

5.4.4 Jumper pins

Two sets of two pins — labelled J1 and J2 — are the final components on the board. They are located in the upper half of the board, between the wide traces connecting coaxial cable connectors to the GPIO header. The upper pins connect to C16 and F16 pins of the De0-Nano, while the bottom ones are connected to the ground plane. The intended use was for the C16 and F16 pins to have pullup resistors enabled. Then I planned to use one of the jumpers for switching between master and slave modes. However, a problem was encountered that led to the jumpers not being used in the final design.

The voltage output by the De0-Nano board is not sufficient for driving the ultrasonic actuators used in [10]. Therefore my colleagues designed a PCB that boosts the outputs of the generator to a higher voltage. However, this booster board interfered with the voltage on those jumpers. When the voltage boosting was turned on, noise would be induced on the jumper pins, causing the device to switch between master and slave modes. Thus I moved the functionality from the synchronisation shield PCB to the switches on the DE0-Nano board itself.

6 | Assessing the generated signals

In this chapter, I will provide measurements of signals generated by the device and compare them with the settings used to generate them in order to assess the accuracy of the device.

6.1 Experimental setup

Two synchronised generators were used to generate the signals. Due to the ongoing Covid-19 pandemic, the experiment had to be undertaken in somewhat improvised conditions — only two DE0-Nano kits were at hand and the synchronisation shield PCBs described in section 5.4 were not available. Thus, instead of using coaxial cables, the De0-Nano boards were connected by means of female to female jumper cables[1].

A Hameg HMO3524 oscilloscope with two HZ350 probes was used to take the measurements. Channel 1 of the oscilloscope (yellow waveform in screenshots) was connected to an output pin of the slave generator, while channel 2 (blue waveform) was connected to an output of the master generator[2].

6.2 Measurements

6.2.1 Measurements at 40 kHz

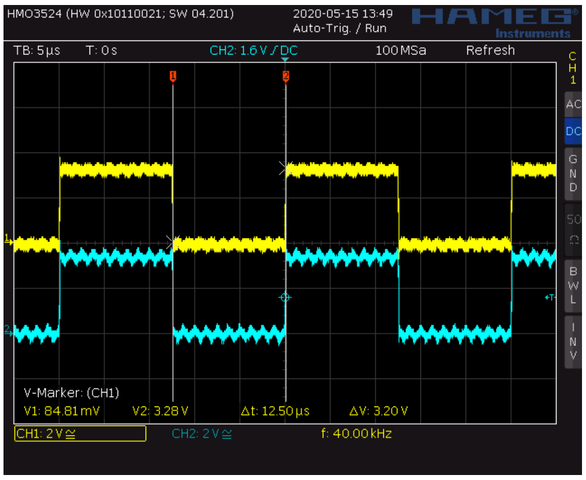
The first set of measurements was taken with the output frequency set to 40 kHz. This corresponds to a period of 25 µs. A single degree (the resolution of the generator) is then equal to $\frac{25}{360}$µs, or about 69.4 µs.

A measurement of the outputs when both their phase shifts are set to zero and their duty cycles to 180° is shown in figure 6.1. Within the figure, the low half of the period is measured as lasting 12.5 µs, which is the correct duration for 180° duty cycle. A detailed view of the rising edge shows that the signals are perfectly synchronised; their phase shift is indeed 0°.

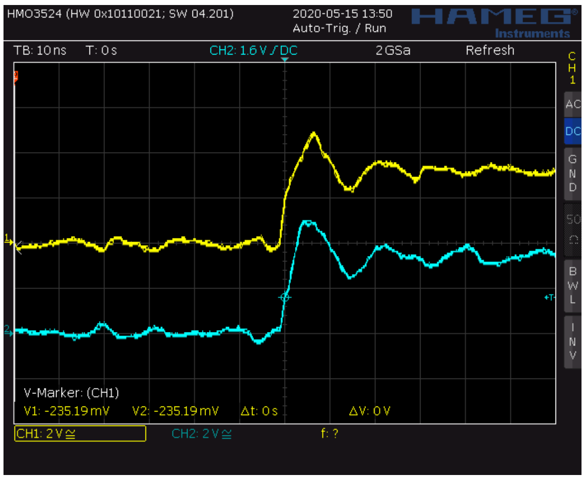
In a second experiment, the duty cycle of both signals is set to 4°. Their phase shift is then increased from 0° to 6°. Two measurements (for 1° and 2°) are shown in figure 6.2.

[1]Cables of equal length were selected for connecting the pins carrying the fast clock.

[2]This information is provided only the sake of completeness. Whether a device is operating in master or slave mode should not have any effect on signals generated by it.

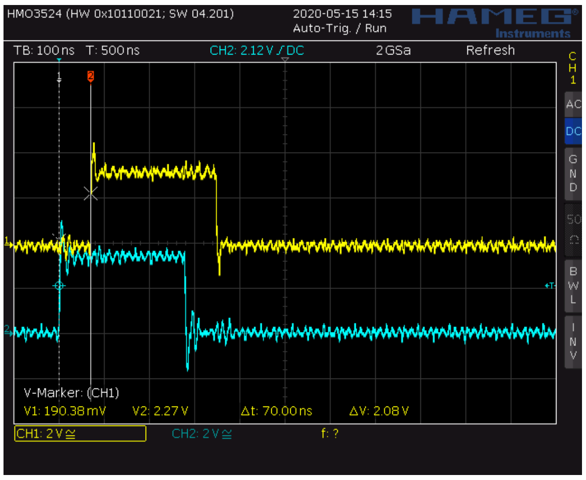


(a) Full duty cycle

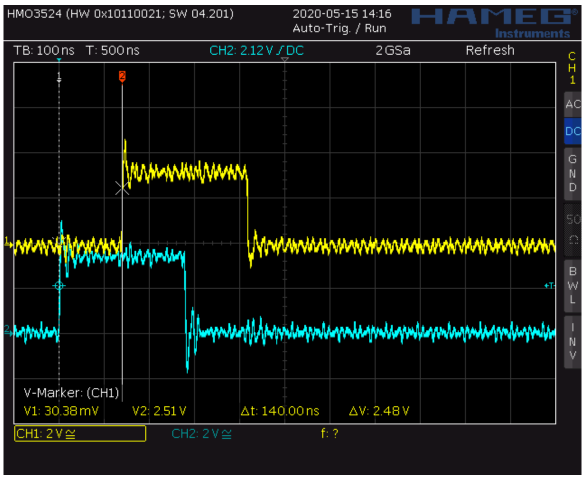


(b) Detail: rising edge

Figure 6.1: Measurement: 40 kHz, 180° duty, 0° phase



(a) 1° phase



(b) 2° phase

Figure 6.2: Measurement: 40 kHz, 4° duty

The values of measured delays for all 6 degrees of phase shift are provided in table 6.1. The width of the pulse was measured as 278.0 ns, with the expected value being 277.6 ns. The value for zero phase shift is omitted, as no delay of rising edge could be measured.

	Rising edge delay		
Set phase [°]	Expected [ns]	Measured [ns]	Difference [%]
1	69.4	70.0	0.86
2	138.8	140.0	0.86
3	208.2	210.0	0.86
4	277.6	278.0	0.14
5	347.0	348.0	0.29
6	416.4	418.0	0.38

Table 6.1: Expected and measured delays of rising edge in 4° pulse experiment at 40 kHz

6.2.2 Measurements at 100 kHz

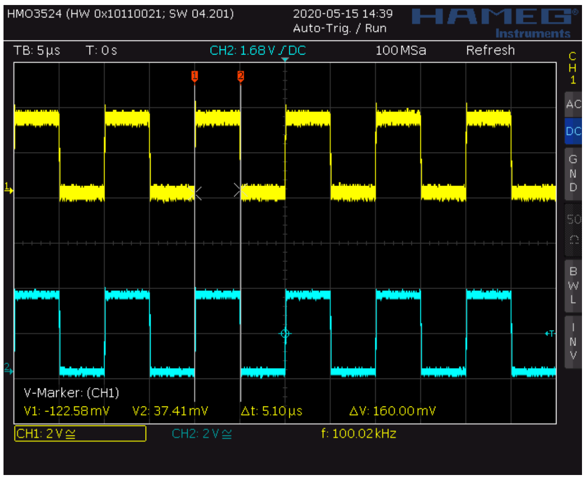
The measurements described in the previous section were repeated for an output frequency of 100 kHz. At this frequency, the period of the signal equals 10 µs, with a single degree corresponding to about 27.8 ns.

Figure 6.3 shows the results for 180° duty and 0° phase. The high half of the wave's period is measured as 5.1 µs, with the expected value being 5.0 µs. This difference might be a measuring error caused by my inappropriate choice of time base. A detail of the rising edge shows that, as was the case with 40 kHz output, the rising edges are synchronised.

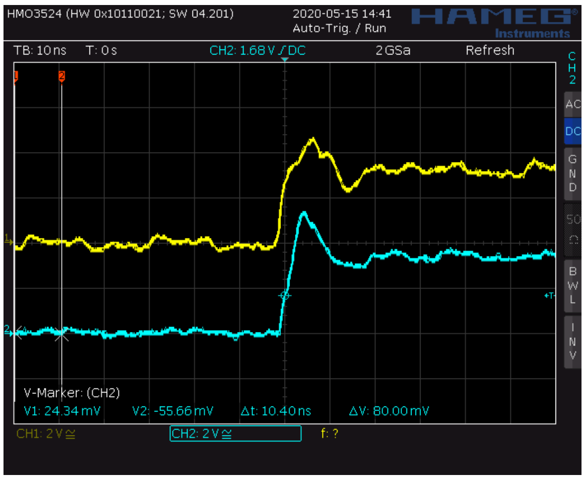
Figure 6.4 shows two measurements of the second experiment, where the duty of both signals is set to 4°, and the phase of the slave is varied from 0° to 6°. Those, as well as the remaining 4 measurements, are provided in table 6.2. The width of the pulse in this experiment was measured as 112 ns, with the expected value being 111.2 ns.

	Rising edge delay		
Set phase [°]	Expected [ns]	Measured [ns]	Difference [%]
1	27.8	28.0	0.72
2	55.6	56.0	0.72
3	83.4	85.0	1.92
4	111.2	112.0	0.72
5	139.0	141.0	1.44
6	166.8	168.0	0.72

Table 6.2: Expected and measured delays of rising edge in 4° pulse experiment at 100 kHz

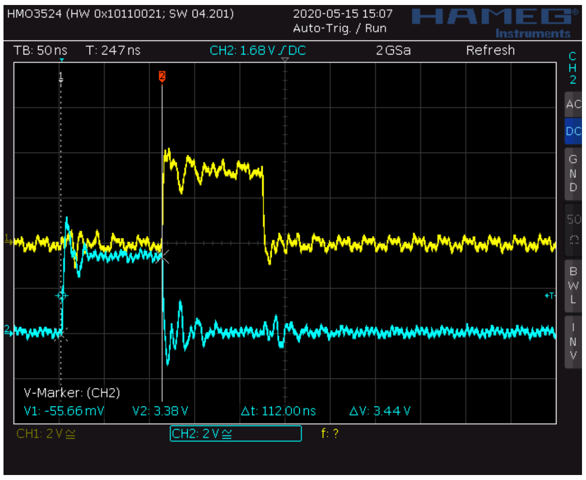


(a) Full duty cycle

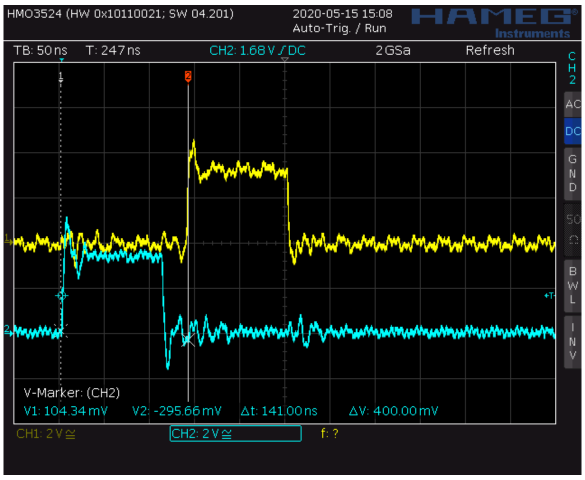


(b) Detail: rising edge

Figure 6.3: Measurement: 100 kHz, 180° duty, 0° phase



(a) 4° phase



(b) 5° phase

Figure 6.4: Measurement: 100 kHz, 4° duty

6.3 Assessing the output frequency range

Due to the unavailability of the synchronisation PCBs, it was not possible to measure the range of achievable output frequencies. In my improvised setup, I have already encountered difficulties after increasing the output frequency to 200 kHz. However, this is not representative of the actual capabilities of the device. The synchronisation shield PCBs with their coaxial cables are surely better suited to the task of transmitting signals with frequencies in the range of tens of MHz[3] than the jumper cables I had to make do with.

Still, some estimation of the frequency range can be made. Clearly, frequencies of at least 100 kHz are achievable, as demonstrated in the previous section. According to the Cyclone device datasheet [1] [page 24], the maximum output frequency of the phase-locked loop of the FPGA used within DE0-Nano is 472.5 MHz. Dividing this by 360, we arrive at 1.3125 MHz, which is the absolute maximum output frequency of the generator.

The actual maximum output frequency will likely be significantly lower than that. I expect it to be limited by the maximum frequency of fast clock which the synchronisation shield PCB and the coaxial cables are able to transmit.

6.4 Conclusions

The generated signals appear sufficiently precise. The worst inaccuracies were observed when measuring the pulse width of 180° duty cycle signal at 100 kHz (measured width 2% greater than expected) and in the phase shift measurement for 3° phase shift in table 6.2 (measured phase shift 1.92% greater than expected). However, since the rising edges of signals with equal phase shifts appear to be perfectly synchronised, I suspect measuring error to be the dominant source of the aforementioned inaccuracies.

It is also within the realm of possibility that the improvised means of connecting the generators worsened their performance.

The maximum output frequency of the generator could not be measured precisely, but it is at least 100 kHz and at most 1.3125 MHz.

[3]The frequency of the logic clock shared among devices is always 360 times higher than the output frequency. For output frequencies of 40, 100 and 200 kHz this works out to be 14.4, 36 and 72 MHz, respectively.

7 | Conclusion

The generator of phase-shifted square waves which I have developed, implemented and then described in this thesis meets all of the requirements set out in section 1.3. The generator is based upon the DE0-Nano FPGA development board and implemented in VHDL. Its source code is made available in a github repository[1].

The generator can operate alone, providing 64 outputs, or synchronised with up to 3 other generators, in which case the number of outputs reaches 256. To synchronise multiple generators, I designed a custom PCB shield, which I describe in section 5.4. To increase the number of channels beyond 256, one would need only to modify this shield by adding more clock output connectors. The VHDL implementation itself is agnostic to the number of synchronised devices and will not need to be modified except in an extreme case, which I discuss in section 5.3.12.

The duty cycle and phase shift of each signal generated by the device is configurable with a resolution of 360 parts per period. The signals are synchronised across devices and accurate, as shown in chapter 6. Additionally, the output frequency shared by all signals can be reconfigured on the fly.

The generator communicates with the user via UART. As discussed in section 4.4, the device is capable of having its duties and phases[2] updated more than 150 times per second, which exceeds the requirement set out by the assignment (50 Hz). In addition, if only phases or only duties are reconfigured, this refresh frequency is doubled.

Serving as further proof that the goals have been met and that the device is fit for its purpose is the fact that throughout the development, it has been successfully utilised in applications by my colleagues. Those are the two applications listed as primary use cases in the introductory chapter of this thesis — dielectrophoresis and acoustic manipulation. A very early version of the generator has been used in bachelor theses of my colleagues [9] [5] to drive 64 actuators in the AcouMan platform. Currently, the generator is being used in an expanded version of said platform to control 256 actuators. A video demonstration of the extended platform in action is available at `https://www.youtube.com/watch?v=D5RClzG8gOU`.

[1]https://github.com/aa4cc/fpga-generator

[2]That is, the duty cycle and phase shift settings of its 64 outputs.

7.1 Ideas for further developement

In this section, I will provide three ideas for further improvement of the design.

7.1.1 Differential transmission of logic clock

When multiple devices are synchronised together, the master device has to provide the logic clock signal to all of them. In my design, this signal is passed over single-wire coaxial cables, with one coaxial cable per device, including the master (see section 5.2 for details). Although the coaxial cable appears to be adequate in my colleagues' application, using a twisted pair cable in its stead and transmitting the logic clock differentially could make the device more resilient to interference.

7.1.2 Eliminating the reset wire by measuring time since last logic clock rising edge

When multiple devices are operating in a synchronised fashion, each derives its own reference clock from the shared logic clock signal by means of a clock divider. However, those clock dividers may become desynchronised, which in turn causes the output signals of individual devices to desynchronise. To combat this, I implemented a procedure that allows the master device to synchronise the dividers by resetting them. This procedure is described at the end of section 5.3.11 and illustrated in figure 5.5. To facilitate the synchronisation, a reset signal was introduced — the reset wire is driven by the master device and connected to all slaves.

However, this reset wire could perhaps be eliminated from the design. Instead, the synchronisation procedure could consist solely of the master device disabling its clock outputs for some sufficiently long time. Each device could then internally measure how much time has elapsed since the last rising edge of logic clock. Upon reaching a threshold, each device would reset its clock divider.

A problem that would have to be tackled is determining how long this threshold should be. Clearly, it would need to be longer than the period of logic clock, since otherwise the dividers would keep constantly resetting themselves and thus cease to function. However, the period of logic clock is dependant on the output frequency of the generator, which may be changed on the fly. Perhaps this could be solved by the device keeping in memory the last observed length of time between rising edges of logic clock and using as the threshold, for example, double this value.

7.1.3 Dynamically reconfigurable resolution for increased frequency range

In my implementation, the generator has a fixed resolution of 360 parts per period. As a result, the frequency of the logic clock, which is shared among the generators, has to be 360 times higher than the output frequency. This is a limiting factor — the frequency of the logic clock cannot be increased infinitely (as discussed in section 6.3). Therefore, there will exist some maximum output frequency achievable by the generator.

However, if the resolution of the generator was lower, higher output frequencies would become possible. For example, if the resolution was decreased to 180 parts per period, the maximum frequency would double. This might be done dynamically by introducing a new command to the communication protocol. This command would carry in its data field a single integer denoting the desired resolution. Upon receiving it, the device would update its clock divider to the new dividing factor. After reconfiguring the phase-locked loop[3], the user could use the generator at their new chosen resolution (taking care to ensure that the integers they use to configure phase shifts and duty cycles do not exceed this new resolution).

Another application of this functionality would be increasing the resolution when the device is used at low output frequencies. The user would not need to limit themselves to a resolution of 360 parts per period if the output frequency of the logic clock was beneath the maximum. The increase of resolution would be limited by the fact that unsigned 9-bit integers are used to represent the duty cycles and phase shifts in the design. Thus, the maximum resolution would be 512 parts per period. It would, of course, be possible to increase the size of used integers or to implement reconfigurable integer sizes. The former would prolong communication times, leading to worse refresh rate, while the latter would involve significant changes to the code.

[3]Note that the frequency of the fast clock generated by the phase-locked loop needs to always be N times higher than the output frequency, where N is the resolution of the generator (and thus also the dividing factor of the clock dividers).

Appendices

A | GPIO pins used by the design

This appendix provides a complete list of pins used by the design. The list is split into two tables — table A.2 provides the mapping of output channel pins. Those pins carry the 64 square wave outputs of the generator.

The other table (A.1) lists the remaining pins used in the design, along with notes on pin configuration where necessary. Within this table, I adhere to the same labelling of pins as I have used previously in figures 5.1 and 5.2.

Label	Pin	Configuration note
UART IN	A8	
UART OUT	D3	
CLK OUT 1	B16	$50\,\Omega$ series termination enabled
CLK OUT 2	D16	$50\,\Omega$ series termination enabled
CLK OUT 3	D14	$50\,\Omega$ series termination enabled
CLK OUT 4	G16	$50\,\Omega$ series termination enabled
CLK IN	E15	
TRIGGER EXTERNAL	G15	Pull-up resistor enabled
RESET BIDIR	F14	Pull-up resistor enabled
CHAINING SWITCH	M1	
MASTER SWITCH	T8	
DEVICE LED	A15	
SHIELD LED	C15	

Table A.1: Non-channel pin mappings

Channel	Pin	Channel	Pin
1	J13	33	D12
2	K15	34	D11
3	J16	35	A12
4	L13	36	B11
5	M10	37	C11
6	N14	38	E10
7	L14	39	E11
8	P14	40	D9
9	N15	41	C9
10	N16	42	E9
11	R14	43	F9
12	P16	44	F8
13	P15	45	E8
14	L15	46	D8
15	R16	47	E7
16	K16	48	E6
17	L16	49	C8
18	N11	50	C6
19	N9	51	A7
20	P9	52	D6
21	N12	53	B7
22	R10	54	A6
23	P11	55	B6
24	R11	56	D5
25	T10	57	A5
26	T11	58	B5
27	R12	59	A4
28	T12	60	B4
29	R13	61	B3
30	T13	62	A3
31	T14	63	A2
32	T15	64	C3

Table A.2: Pin mappings of generator output channels

References

[1] Cyclone iv device datasheet. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyiv-53001.pdf, 2016.

[2] Cyclone iv device handbook. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf, 2016.

[3] Altpll (phase-locked loop) ip core user guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_altpll.pdf, 2017.

[4] T. Corkett A. Marzo and B. W. Drinkwater. Ultraino: An open phased-array system for narrowband airborne ultrasound transmission. *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 65(1):102–111, January 2018.

[5] Kollarčík Adam. Planar acoustic manipulation with spherical objects on solid surface. *Bachelor's thesis*. http://hdl.handle.net/10467/76141, 2018.

[6] William Beasley, Brenda Gatusch, Daniel Connolly-Taylor, Chenyuan Teng, Asier Marzo, and Jose Nunez-Yanez. High-Performance Ultrasonic Levitation with FPGA-based Phased Arrays. *arXiv e-prints*, page arXiv:1901.07317, January 2019.

[7] Daniel Connolly-Taylor, William Beasley, Brenda Gatush, Asier Marzo, and Jose Nunez-Yanez. Ultrasonic levitation with phased arrays and the xilinx zynq platform. https://github.com/eejlny/ultrasonic-levitation-with-Xilinx-Zynq/blob/master/doc/long_report.pdf.

[8] T. Michálek J. Zemánek and Z. Hurák. Phase-shift feedback control for dielectrophoretic micromanipulation. *Lab on a Chip*, 18(12):1793–1801, 2018.

[9] Matouš Josef. Manipulation with objects on a surface of a liquid using an array of ultrasonic transducers. *Bachelor's thesis*. http://hdl.handle.net/10467/76751, 2018.

[10] Josef Matouš, Adam Kollarčík, Martin Gurtner, Tomáš Michálek, and Zdeněk Hurák. Optimization-based feedback manipulation through an array of ultrasonic transducers. *IFAC-PapersOnLine*, 52(15):483–488, 2019.