PALACKÝ UNIVERSITY OLOMOUC
FACULTY OF SCIENCE

Department of Experimental Physics

Arbitrary digital sequence generator for controlling photonic experiments

BACHELOR THESIS

Radim Hošák

2016

PALACKÝ UNIVERSITY OLOMOUC
FACULTY OF SCIENCE

Department of Experimental Physics

Arbitrary digital sequence generator for controlling photonic experiments

BACHELOR THESIS

Author: Radim Hošák

Study programme: B1701 Physics

Field of study: Applied physics

Form of study: Full-time

Supervisor: Mgr. Miroslav Ježek, Ph.D.

Thesis submitted on: .....

UNIVERZITA PALACKÉHO V OLOMOUCI
PŘÍRODOVĚDECKÁ FAKULTA

Katedra experimentální fyziky

Generátor digitálních pulzních sekvencí pro řízení optických experimentů

BAKALÁŘSKÁ PRÁCE

Vypracoval: Radim Hošák

Studijní program: B1701 Fyzika

Studijní obor: Aplikovaná fyzika

Forma studia: prezenční

Vedoucí bakalářské práce: Mgr. Miroslav Ježek, Ph.D.

Práce odevzdána dne: .....

Abstract

With the rapid development of microcontroller units (MCU), MCU-based devices for digital experiment control are being deployed in a wide range of experiments. In this Thesis, a concept of a multi-channel digital pulse sequence generator is designed, followed by a prototype of such a device. Characteristics of the resulting device are measured and its functionality is demonstrated in a stopped-light experiment.

Keywords

microcontroller, electronic pulse generator, experiment control

Acknowledgement

I wish to express my gratitude to my supervisor, Miroslav Ježek, who patiently guided me throughout the work. His insight and advice helped me when tackling many obstacles along the way. I am also very thankful to everyone from the Quantum Optics Lab Olomouc for their support.

Declaration

I hereby declare that I myself wrote this bachelor thesis under the supervision of Miroslav Ježek while using the resources listed in the References.

Signed in Olomouc on .....

Contents

1 Introduction

In almost any modern optical measurement or experiment there is a need for numerous electronical components to provide vital functionality such as optical or radio-frequency (RF) signal generation, pulse sequencing, and switching various electronical devices. Another area of interest for optical applications is experiment control, for example laser stabilization or locking to a resonant frequency of a cavity. Data acquisition is also a vital part of optical setups as measured data need to be stored, post-processed or used for feedback or feed-forward.

These challenges have been typically overcome with electronical circuitry working with analog signals. A great downside to these analog-based solutions is that their scalability and upgradability is cumbersome. Even when the upgrade is succesfully performed, one often ends up with a convoluted electronical circuit which will most likely be hard to maintain in the future. On the other hand, analog-based solutions are known for their low-noise operation.

In the last decades the technology to achieve the same results using digital devices was becoming available in the form of application specific integrated circuits (ASIC) or user-reconfigurable complex programmable logic devices (CPLD) and field programmable gate arrays (FPGA). The emerging microcontroller units (MCU) could, in principle, make the whole development process much simpler and faster, as they are easily configured using the standard C programming language and come with many useful peripherals, but for a long time their computing power and clock frequency was not high enough to allow them to be used in computation-heavy applications or situations where precise timing was critical.

The situation has been changing lately, as more and more capable MCUs, often used as a part of hobbyist-grade development boards, are now in the centre of in-house developed and specifically tailored solutions. Experiments and often entire labs are then not dependent on a commercially available specialized product which often does not meet the experimentalist's needs and is not extensible beyond its planned functionality. With a powerful MCU, one has got a universal device with a broad spectrum of possible applications. Modern MCUs offer numerous peripherals, such as analog-to-digital converters (ADC), digital-to-analog converters (DAC) or hardware number generators (RNG), which can be used as a crucial part of experimental setups where there is a need for, for example, measurement of an analog voltage input, analog output or randomization, respectively. Most MCUs are capable of interfacing with the outside world via standard communication protocols, such as I^2C, SPI, USB or Ethernet. When the capabilities of an MCU are insufficient, they can often be extended with add-on boards, basic external circuitry or so-called "shields" which are connected on top of an MCU development board.

A great advantage of MCUs over analog-based solutions is that their functionality is software-driven. This means that modifications and upgrades of this functionality are achieved simply by modifying the source code. This way, the experiment can grow and change according to the experimenter's needs very quickly, compared to analog-based electronical circuitry, which always has to be manually built according to a certain schematic. Furthermore, some development boards can be readily accesed and controlled remotely from a computer using a high-level programming interface, such as Matlab or LabVIEW, making the development process even easier and faster.

Despite many succesful implementations of an MCU one must not forget the limitations that necessarily come with they employment. Firstly, when precise timing is needed, for example in event sequencing, synchronization or data acquisition scenarios, one must take into consideration that the minimum possible time uncertainities achieved with an MCU will be limited by its clock rate and clock precision. It must however be noted that the

clock speed of the MCUs is improving fast and many algorithms might be optimized for speed by being implemented in assembly language (asm). Another caveat of digital-based solutions in general is the increase in noise compared to analog-based solutions. These downsides might or might not be an issue, depending on the application.

The suitability of MCUs in experimental and measuring applications has been repeatedly proven by its succesful employment in a broad range of experiments. One of the most common use cases is active stabilization of an experiment, where a control system is required to acquire error signals and use them for feedback. Stabilization of laser frequency is often done by locking to a resonant frequence of a Fabry-Perot cavity using for example the Pound-Drever-Hall (PDH) technique. These processes often incorporate feedback and proportional-integral-derivative (PID) controllers. MCUs have also been used instead of analog-based feedback loops and algorithms for automatical relocking have been implemented for PDH cavity stabilization of a fiber laser [1] and in various experimental scenarios, such as optical tweezers [2], Raman memory [3] and squeezed states [4]. Usage of an MCU for stabilization of a Mach-Zehnder interferometer has also been succesfully demonstrate [5].

An MCU has also been used in atomic force microscopy for driving DC motors in an experimental fluid-injection system [6].

MCUs are also starting to be used in data acquisition (DAQ). A DAQ system for a silicon photomultiplier (SiPM) detector has been built [7] using a custom add-on shield for a commercially available MCU development board and an accompanying mobile application for the Android operating system. An MCU-based solution has also been implemented for data acquisition from SiPM detectors during high-altitude balloon flights [8]. This scenario shows the ease of extension of the MCU's functionality with 3G and GPS modules and fulfilling of low-pover operation and low-cost requirements.

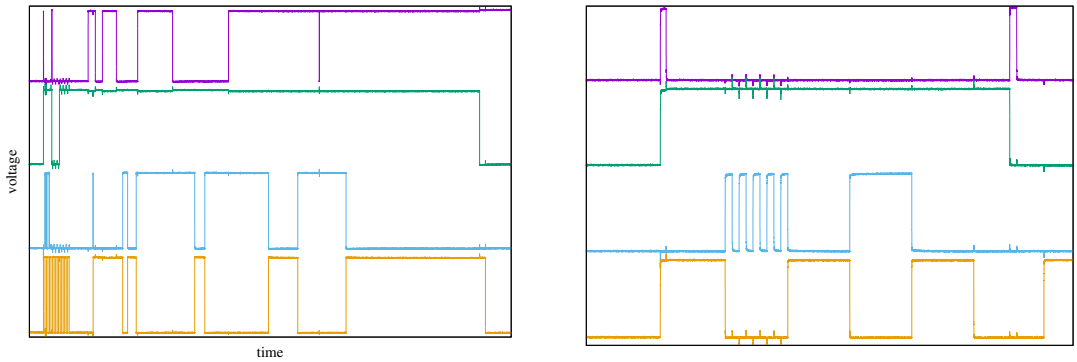
The usage of an MCU for a true random number generator based on a pseudo-random number generator (PRNG) seeded with supposedly random input from unconnected analog pins of the MCU has been tested [9]. It has, however, been found that the output of a random number generator realized in this manner yielded too low entropy for a sensible cryptographical application. On the other hand, it should be noted that many modern MCUs come equipped with a built-in hardware true random number generator (TRNG). These hardware random number generators often pass the standardized test suites, such as the NIST Statistical Test Suite.

This Thesis is focused on the application of an MCU as an event sequencer. Generally speaking, these devices accept as an input a sequence of events to be executed at a given time with as high temporal precision as possible. An event is merely a change in the output of the sequencer and can be digital (one-bit event sequencer) or analog (with as many bits as the DAC used). Event sequencers can optionally work with multiple output channels in parallel. This devices have been designed for experiment control using an MCU [10, 11, 12] and also an FPGA [13, 14].

In the Quantum Optics Lab at the Department of Optics of the Palacký University, there are many experimental challenges which would benefit from a multi-channel digital event sequencer with digital and analog outputs, namely stabilization of laser wavelengths, driving acousto-optic and electro-optic modulators (AOM and EOM, respectively), variable-gain amplifiers (VGA), direct digital synthetizers (DDS), electronic switches, and synchronization and/or triggering of various parts of the experiment. The one-bit output signal of this sequencer is expected to assume the form of two alternating voltage levels corresponding to logical `true` and `false`. This alternation of voltage creates a sequence of pulses and hence I will further refer to our device as an arbitrary digital pulse generator, or simply *pulsebox*.

It is required of the device to be equipped with at least 8 one-bit digital channels for simultaneously outputting the sequences. The shortest pulse one could generate using the pulse box should not be much longer than 100 ns, which is also our granularity requirement. This means that we want to be capable of generating pulses which differ in length by 100 ns. There should be very low uncertainity (jitter) the timing of the sequence—when the sequence is repeated multiple times, no changes in its shape should be observed. Also, to facilitate sequence generation, the device should be remotely programmable and have a basic capability of automated operation made possible with scripts.

The reason why commercially available function generators cannot be used in place of the newly designed device is *sparse sequence generation*. In most of our use cases, the sequences will be made of changes of the digital output with great temporal spacing. Due to the large ratio between the time intervals when the digital output is stationary and the time intervals during which it changes, the sequences cannot be easily generated even by the function generators equipped with a programming interface and an internal memory for arbitrary waveforms.



(a) An example of a complicated sequence. **(b)** The beginning of the sequence (in detail).

Figure 1: An example of a complicated multi-channel pulse sequence (a) which cannot be easily generated using a conventional function generator. The beginning of the sequence (b) is shown in detail. This sequence has, in fact, been generated by the prototype of the device described in this Thesis.

A computer-based solution in conjunction with a standard serial or parallel interface cannot be used because of its large jitter and insufficient granularity. An implementation of a pulsebox using a computer sending events—information about the length of pulses in the sequence—to an MCU as the sequence was being generated has been proposed, but it has been found that the serial communication used was accompanied by interrupts of the MCU's processor, which resulted in significant jitter.

As our requirements for this device are quite specific and we require a great deal of flexibility in terms of its operation, there was no other way but to build a custom in-house solution. It was proposed by my supervisor that the solution be MCU-based. Using an FPGA was also a possibility, but there is no need to implement this kind of a solution, as our jitter and granularity requirements can be satisfied using an MCU with a reasonably fast clock.

My first goal was to create a conceptual design of the pulsebox along with a general description of the device, its operation and means of programmability and automatization. A pulsebox prototype was then to be built. Lastly, the characteristics of the device were to be listed, measured and calibrated using calibration procedures, which were to be designed with further automatization in mind.

2 A general description

The primary functionality of the presented MCU-driven pulsebox is to generate multiple digital pulse sequences. The pulsebox is designed to be running in two modes—triggered and continuous. The triggered mode runs a sequence as an interrupt routine, meaning that it waits for a rising edge of a trigger signal on a dedicated channel. The continuous mode repeats the sequence with a given frequency. It is not as sophisticated as the triggered mode, but it is sufficient for applications where no synchronization with outer parts of the experimental setup is needed.

For each of the pulsebox sequences there is one digital output pin of the MCU. A pulse sequence on one channel is achieved by precisely timed switching of the output of the corresponding pin's state to a logical `true` or `false`, or `HIGH` or `LOW`, respectively. Sequence generation occurs simultaneously on all digital channels of the pulsebox, which means that all the sequences run in parallel. From now on, the term sequence will be used to refer to the ensemble of all the sequences running on the pulsebox channels. For further upgradability and scalability, sixteen pulsebox channels (`CH1` through `CH16`) have been implemented. They are all equal and can be used arbitrarily, but I reckon that the last eight channels are going to be used in conjunction with a DAC or VGA with parallel inputs. It is for this reason that I will sometimes refer to them as parallel output channels (`CHP`).

USB connection is used to provide power to the MCU and the entire pulsebox device. USB is also used as a means of a one-way communication with the pulsebox. It is used only for programming the MCU. The end user inputs the parameters of the multi-channel sequence to a computer program, which then generates the corresponding source code. The source code is then compiled and uploaded to the memory of the MCU.

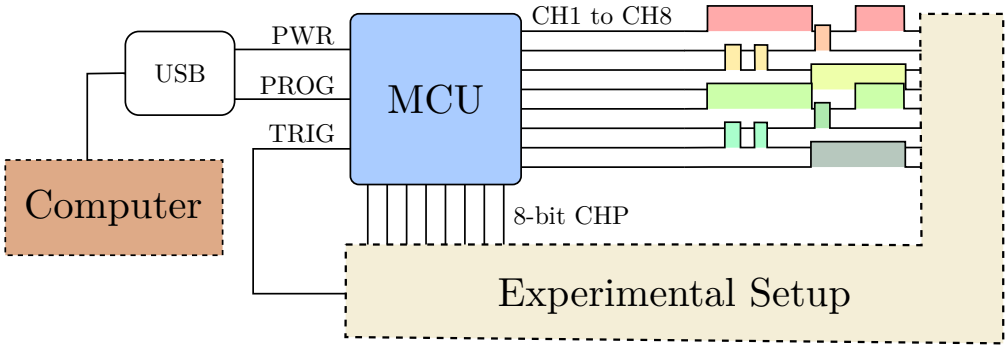


Figure 2: A conceptual diagram of the pulsebox. The MCU is powered and programmed via a USB connection. Sixteen pins are used as pulsebox channels—eight arbitrary pulsebox channels (`CH1` to `CH8`) and eight parallel channels (`CHP`). The parallel channels differ in no way from the arbitrary channels and are just a proposition of a specific channel set for future applications. An example of a sequence is shown on the arbitrary channels. The resulting sequence is fed into the experimental setup.

2.1 Basic concepts

We can call any change of the pulsebox output (such as the beginning or the end of a pulse) by the term *event*, as it is bound to a physical change in the operation of the experimental setup. The output change event is given by a 16 bit wide binary number that specifies which channel is to be `HIGH` and which `LOW`. It will also prove useful to refer to an event even in the case of a delay between two such changes of output. See Figure 3 for a visualization.

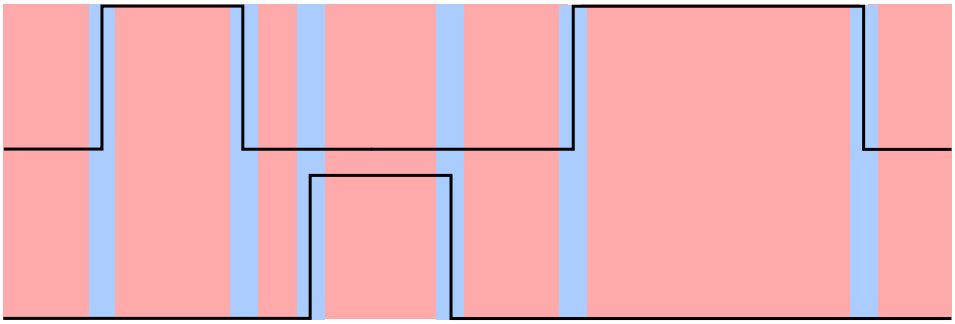


Figure 3: A visualization of the low-level pulsebox events on a two-channel sequence. The blue parts of the sequence are the output change events, while the red parts are delay events.

While the two low-level pulsebox events—*output change* and *delay*—are universal enough to allow for the creation of all the desired sequences and pulse patterns, it is desirable to have a virtual layer of high-level events. For example, there are many scenarios where parallel communication with some other electronic device is needed. For this purpose, the *parallel output* events have been created. Instead of having to convert the desired decimal value to binary and then redistributing each bit of the result to a particular pulsebox channel, *parallel output* allows the end user to specify the decimal value itself. The rest is taken care of by the pulsebox software. On the other hand, if the end user wishes to only change one channel of the pulsebox, it should not be required of him to specify the states of the remaining channels. Thus, the *channel change* event has been implemented. Now, the user only needs to know the target channel and its new state. Even further simplification to the process can be made. When a user demands a change of the state of the channel, it will in most cases be a *channel flip* event, which means that the channel changes its state to the opposite of its previous state. Only the target channel number then needs to be known in order to uniquely specify this event.

When a need arises for another high-level pulsebox event, it can be readily implemented using the low-level events, or even the existing high-level events. The entire spectrum of pulsebox events as of writing this Thesis is shown in Table 1.

	event	description	parameters
low-level	output change	changes the value of all the pulsebox channels at once	a 16-bit binary number indicating which channels are `HIGH` and `LOW`
	delay	waits for a predefined amount of time	a time interval
high-level	channel set	sets a given channel to a specified value	channel number, channel state (`HIGH` or `LOW`)
	channel flip	changes the state of a channel to its current opposite	channel number
	parallel output	outputs a given value represented in binary using the parallel pulsebox channels (`CHP`)	a decimal value to be converted to binary

Table 1: An overview of low-level and high-level pulsebox events.

Any imaginable pulse sequence can be represented by a list of events. This list will be called the *event queue* from now on. The sequence is uniquely given by a list of

output change events along with their timestamps—the times at which the output change events occur. An event queue consisting solely of a list of output change events with their timestamps will be called a *bare event queue*. As our concept of the pulsebox is built upon the idea of an MCU alternating between output changes *and* delay loops, there is also a need to include all delay events in the event queue in order to create the desired spacing between events. The resulting event list is called a *full event queue*. A full event queue can be built from a bare event queue by simply subtracting the timestamps of any two neighbouring events. In practice, the process of generating a sequence begins with the end user, assisted by the accompanying pulsebox software, specifying all output change events and their timestamps, giving the pulsebox software a bare event queue representing the desired sequence. The software then automatically converts the bare event queue to a full event queue.

While it might be possible to communicate with the pulsebox via a serial interface and send commands and event requests to it during its operation, it would introduce unwanted delays and jitter in the sequence timing. Serial communication requires MCU interrupts, which cannot, in principle, be entirely eliminated. As the interrupts might come on seemingly random occasions, it is also impossible to adjust the sequence to correct for them. Using pre-generated source code rids us of this trouble. It is, however, only applicable in situations where the sequence is known *a priori*. When the sequence has to be generated on-the-go, either serial communication or clever pre-programming of the MCU is needed. While the latter solution seems attractive, its chance of success depends entirely on the specific requirements on the sequence. For example, when random delay pattern is needed, one might take advantage of the built-in TRNG of the MCU. On the other hand, applications where real time sequence adjustment from the outside is the key factor will have to go with some kind of communication.

When an event queue is given, it is possible to generate a source code file appropriate for the given sequence. This procedure can be facilitated if we introduce the concept of *code blocks*. A code block is a piece of source code with one specific effect on the pulsebox operation. Any event in the event queue has got a corresponding code block. For output changes, it is simply an instruction for the MCU to change a value of one of its internal registers. For delays, there are *delay loops* with various number of passes. These code blocks have variable parameters and are thus called *dynamic code blocks*. There is, however, always a need for code blocks that can contain an initialization routine or the closing parentheses at the end of the code. These blocks are always present and are the same for any sequence. They are therefore called *static code blocks* and are most often found in the beginning or at the end of a final source code file. The actual form of the code blocks is shown in Section 2.2. The process of source code assembly starts with creating a text file with an opening static code block containing initialization. Then, events in the event queue are processed in the chronological order. A dynamic code block is generated based on the kind of event and its parameters and it is then appended to the text file. When all events in the queue have been processed, the closing static code block containing closing brackets is added to the end of the file.

In order to create a certain pulse sequence using our pulsebox, it is necessary to create the appropriate source code file, compile it, and then upload the resulting machine code to the pulsebox. To facilitate this whole process for the end user of the device, I extended the conceptual design of my pulsebox by a software suite with multiple levels of computer code. On the lowest level, there is a piece of software providing elementary support for all the vital pulsebox functionality. From now on, it will be reffered to as the *pulsebox software module* or just *module* when no misunderstanding can arise. Using this module, it is possible to create a sequence containing only elementary pulsebox events—output changes and delays. From here it is also possible to generate the apropriate source code

file and upload it to the connected pulsebox MCU. Another crucial part of the module is a definiton of a set of pulsebox related errors, such as specifying a channel number of a non-existing channel when adding an event. However, generating warnings for the end user in a case of such an error is not a responsibility of the module. However, in a real application scenario, it would be quite cumbersome to work with such a limited tool. The real purpose of the module is to be able to provide a solid foundation for a higher level of software—one we will reffer to as *user interfaces*. When designing this layer, the main focus was on making it as user-friendly and powerful at the same time as possible. Support for the creation of higher-level pulsebox events has been implemented and other functionality has been added. For example, the end user is capable of making changes to the sequence or save generated sequence data and/or entire generated source files for later use. A crucial feature of user interfaces is giving warnings to the user in case of an error. Two forms of a user interface can be created—a command-line interface (CLI) and a graphical user interface (GUI).

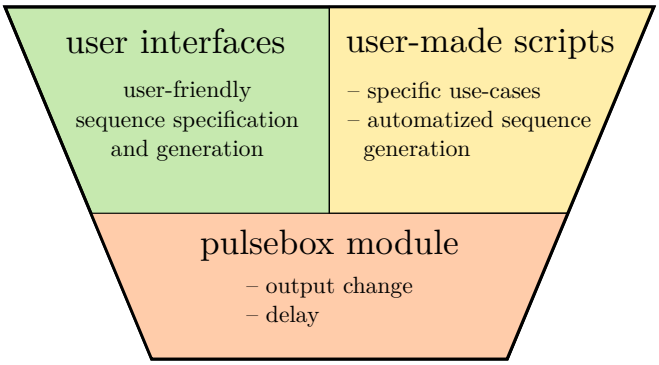
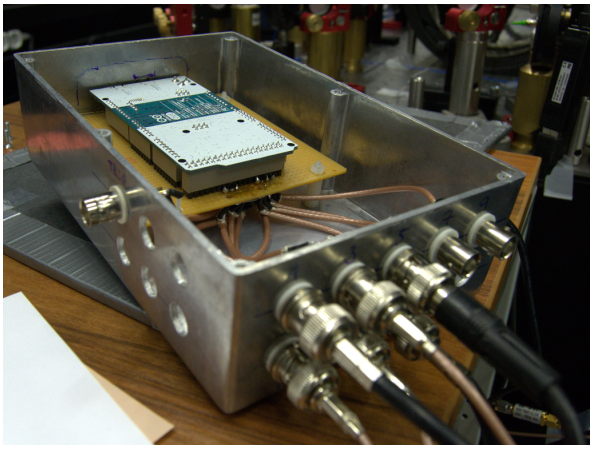


Figure 4: A diagram showing the structure of the pulsebox software. The pulsebox module provides the low-level functionality, while the user interfaces provide high-level pulsebox events and easy ways to create the desired sequence.

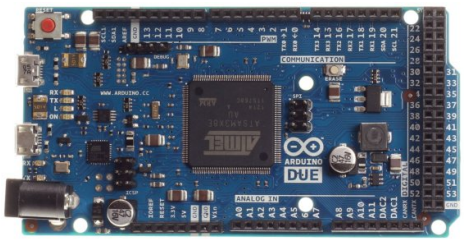
A careful choice of what features are included in the pulsebox module gives us great room for growth in terms of available high-level functionality. For example, routines for automated calibration can be easily implemented using the low-level methods for sequence generation, code assembly, compilation and upload in the form of a script. Along with routines for data acquisition and processing using a multi-channel time-to-digital converter (TDC), crucial calibration data can be retrieved. Another possibility of using these features is adaptive sequence correction. When precise timing is critical, the sequence can be generated and then measured using the TDC. Discrepancies between the user input and the realization of the sequence will then be known, and it will become possible to correct the sequence in a way that brings the end result closer to the desired one.

2.2 The physical realization

The concept of the pulsebox has been brought to life with the use of the Arduino Due development board with a 32-bit Atmel SAM3X8E MCU based on the ARM Cortex-M3 architecture. The MCU's internal clock runs at 84 MHz which provides us with the ultimate granularity limit of roughly 12 ns, which is completely sufficient for our applications. Arduino Due has 54 digital 3.3 V input/output pins, 12 analog inputs, 2 DACs and 4 hardware serial ports. The number of available digital pins is high enough to allow for the implementation of a high number of pulsebox channels. However, it must be taken into consideration that some pins have alternate functions and reconfiguring them for digital input or output will make it impossible for us to use these alternate functions should we wish to do so. The board is equipped with a power jack, which enables us the



(a) The pulsebox prototype.



(b) Arduino Due.

Figure 5: The pulsebox prototype (a) using the Arduino Due development board (b). The cutouts in the metal housing of the prototype are prepared for additional BNC connectors for the pulsebox channels which were not needed in the prototyping phase of the development.

use of pulsebox without the USB connection.

As a base of our prototype (shown in Figure 5) we chose a plain electronics prototyping board. To allow for the Arduino Due development board to be detachable, we equipped the prototyping board with dummy male pin headers which fit their female counterparts on the Arduino board and hold the board in place. Our pulsebox was designed to operate with sixteen digital channels and one channel for external triggering. This means that seventeen physical pins on the Arduino board were selected in order to allow for this functionality. For easier connectivity, a coaxial cable ending with a female BNC connector was soldered to those of the seventeen designated Arduino pins that were used during the prototyping stage of the build. The prototyping board along with the Arduino sitting upon it is housed in a metal box with cutouts for all sixteen digital channels of the pulsebox, the trigger channel and the USB connector for the Arduino board.

The programming language used for the implementation of the pulsebox software suite is Python. It provides fast code prototyping and advanced high-level features that make development easier. The low-level pulsebox module was realized using a custom Python module and a graphical user interface was programmed with `GTK+` bindings for python (the `pygtk` module).

The output change low-level pulsebox event is implemented by changing a value of the Parallel Input/Output's Output Data Status Register (`PIO_ODSR`). The digital pins of the MCU are distributed into four *ports*—`PORTA` through `PORTD`. Each port has its own set of registers, including the `ODSR`. To make sure that all pulsebox channels can be set by changing the register value for just one pin port, all of the pulsebox channels were assigned to the physical pins of the MCU in such a manner that they all belonged to the same port, namely `PORTC`. The `PORTC_ODSR` register is 32-bit wide, with each bit specifying the state of all pins in the `PORTC` port. The most significant bit (MSB) of the register corresponds to the 31^{th} pin of `PORTC`, while the least significant bit (LSB) corresponds to the 0^{th} pin. As the physical layout of these pins on the Arduino board does not adhere to the numerical ordering of the port's pins and the pulsebox channels were chosen so that their corresponding pins were located in one group (see Figure 6), the pin assignement table shown in Table 2 appears to be arranged in an unorderly fashion. However, the location of the pins was a priority for my design. The actual assignment of pulsebox channels to the `PORTC` pins is easily handled by the pulsebox software (see Listing 1).

Figure 6: The physical layout of the pins selected as pulsebox channels. The red pins are the pulsebox output channels of a corresponding number (`CH1` through `CH16`). The purple pin is the trigger channel.

`PIOC_ODSR` bit	`PORTC` pin	pulsebox channel
1	`P1`	1
2	`P2`	10
3	`P3`	2
4	`P4`	11
5	`P5`	3
6	`P6`	12
7	`P7`	4
8	`P8`	13
9	`P9`	5
12	`P12`	9
14	`P14`	8
15	`P15`	16
16	`P16`	7
17	`P17`	15
18	`P18`	6
19	`P19`	14

Table 2: The assignement of the `PORTC` pins of SAM3X8E to the pulsebox channels. Only the pins which were assigned to a pulsebox channel are shown.

```
PINS = (1, 3, 5, 7, 9, 18, 16, 14,\
        12, 2, 4, 6, 8, 19, 17, 15)
def channel2pin(channel):
    return PINS[channel - 1]
```

Listing 1: Assignment of a pin to a pulsebox channel. First, a list of `PORTC` pin numbers asigned to pulsebox channels ordered from `CH1` to `CH16` is defined. The `channel2pin` function returns a pin number from the list which was at the position corresponding to the given channel. [1]

As each PORTC_ODSR register bit controls one PORTC pin and in the general case of an output change event we change all the pins corresponding to the sixteen pulsebox output channels, a true or false values must be written to sixteen PORTC_ODSR bits. An algorithm to achieve this is shown in Listing 2.

```
def comp_odsr(high_channels):
    high_pins = map(channel2pin, high_channels)
    odsr = 0
    for bit in high_pins:
        odsr ^= 1 << bit
    return odsr
```

Listing 2: The computation of the ODSR value. In principle, only the channels which will be set to HIGH are needed to compute the value of ODSR. A logical true is written to the correct positions of ODSR.

Now the code block corresponding to the output change event can be shown (Listing 3).

```
REG_PIOC_ODSR = 0b11111101001111111110; /* all HIGH */
REG_PIOC_ODSR = 0b00000000000000000000; /* all LOW */
REG_PIOC_ODSR = 0b00000000000000000010; /* CH1 HIGH */
```

Listing 3: The code block corresponding to the output change pulsebox events. Three examples are shown. The first sets all pulsebox channel to HIGH, the second sets them all to LOW and the third only sets CH1 to HIGH.

The delay pulsebox event is realized by a *delay loop*. A delay loop is a code structure which does not change the output of the pulsebox. It is used between output changes to provide the desired delay between output change events. The basic idea is that the MCU goes through the loop with a given number of repetitions, more repetitions meaning longer delay. The first realization of a delay loop was made using a standard for loop of the C language (see Listing 4).

```
uint32_t iterations = 10;
uint32_t i;
for(i=0;i<10;i++){
  ;
}
```

Listing 4: A delay loop built with a C for loop. The semicolon inside the loop effectively does nothing, but this "empty" loop still takes time to iterate over. Alternatively, the NOP asm instruction can be used inside the loop.

I found that when the delay loop is implemented in this manner and no modifications are made to the process of its compilation, the compiler simplifies the code, because the delay loop is effectively empty. To avoid this, it is important that the compiler optimizations flags for the arm-eabi-none-gcc compiler are set to -O0. Setting this flag turns off optimizations of the code.

In the end the approach using a C for loop has not been chosen, as the results of pulsebox measurements (which are described in detail in Section 3) did not meet our

requirements. However, the idea of turning off code optimizations remained relevant. As we want to have full control over the timing of a sequence, we cannot allow the compiler to change the structure of any delay loop. The final variant of the delay loop uses code written in inline assembly (see Listing 5).

```
MOVW R1, #0x1
MOVT R1, #0x0
LOOP1:
  NOP
  SUB R1, #1
  CMP R1, #0
  BNE LOOP1
```

Listing 5: A delay loop built with an inline asm loop. First the number of iterations is given. A loop is then started where the iteration number is decremented and compared to zero. The `NOP` instruction inside the loop effectively waits for one clock cycle. The loop repeats as long as the iteration number does not reach zero. The loop is then exited and the execution of the rest of the C code follows. Note that the number of iterations is given by a 32-bit number. Due to the design of the MCU architecture, it must be stored in two steps. First, the lower half-ford (16 bits) is saved, followed by the upper half-word. The loop label is in the format `LOOP#`, where `#` is a number uniquely identifying the given loop throughout the entire C source code.

To assemble the entire source code file, a full event queue is needed. The end user specifies the sequence by a bare event queue—a list output change events with their timestamps. During code assembly, the bare event queue is converted to a full event queue, which contains the corresponding delay events between the output change events. A simplified algorithm that achieves this is shown in Listing 6.

```
def bare_to_full(beq):
    beq.sort(key=attrgetter("timestamp"))
    current_timestamp = 0  # start from zero
    new_delay = DelayEvent()
    full_event_queue = []
    for output_event in beq:
        delay_length = output_event.timestamp\
                       - current_timestamp
        new_delay.length = delay_length
        full_event_queue.append(new_delay)
        full_event_queue.append(output_event)
        current_timestamp = output_event.timestamp
    return full_event_queue
```

Listing 6: Conversion of a bare event queue to a full event queue. The conversion is done by placing delay events between consecutive output change events. The bare event queue (abbreviated as `beq` in this listing) is sorted in the order of increasing timestamp of its output events. The full event queue starts as an empty list. The algorithm iterates over all `output_events` of the bare event queue. The timestamp of the last processed output event (or zero if first output event is currently being processed) is subtracted from the timestamp of the `output_event`. This difference is the delay length of the `new_delay` event, which is appended to the full event queue. The `output_event` itself is then also appended. The result is a full event queue.

Now, when the full event queue is available and all low-level pulsebox events have their corresponding code blocks, the entire source code for the desired sequence can be assembled from the full event queue using an algorithm similar to the one in Listing 7.

```
def assemble_code(full_event_queue, filename):
    with open(filename, "w") as f:
        f.write(CODE_HEAD)
        for event in full_event_queue:
            if event.event_type == "delay":
                code_block = generate_delay(event.time)
            if event.event_type == "output_change":
                code_block = generate_output(event.value)
            f.write(code_block)
        f.write(CODE_TAIL)
```

Listing 7: The source code assembly procedure. First, a static code block with code initialization (`CODE_HEAD`) is written to the file. The event queue is then iterated over all its member events, the type of the event is recognized and a corresponding code block is created using the event parameter in correspondence with Table 1. Finally, a static code block with termination of the code (`CODE_TAIL`) is appended to the source code file, completing the process of source code assembly.

The assembled source code is saved as a `.ino` file, which is a standard source code file format for the Arduino Due. The code is compiled and then uploaded to the memory of the MCU with the command shown in Listing 8.

```
arduino --board arduino:sam:due --port ${ARDUINO_PORT}\
--upload ${INO_PATH}
```

Listing 8: The code upload command. The `ARDUINO_PORT` variable specifies the location of the port used for upload (e.g. `/dev/ttyACM0` for Linux or `COM1` for Windows). The `INO_PATH` variable holds the location of the `.ino` source code file.

The GUI I designed for the pulsebox using the `Glade` user interface designer for `GTK+` in conjunction with the `pygtk` Python module is shown in Figure 7.

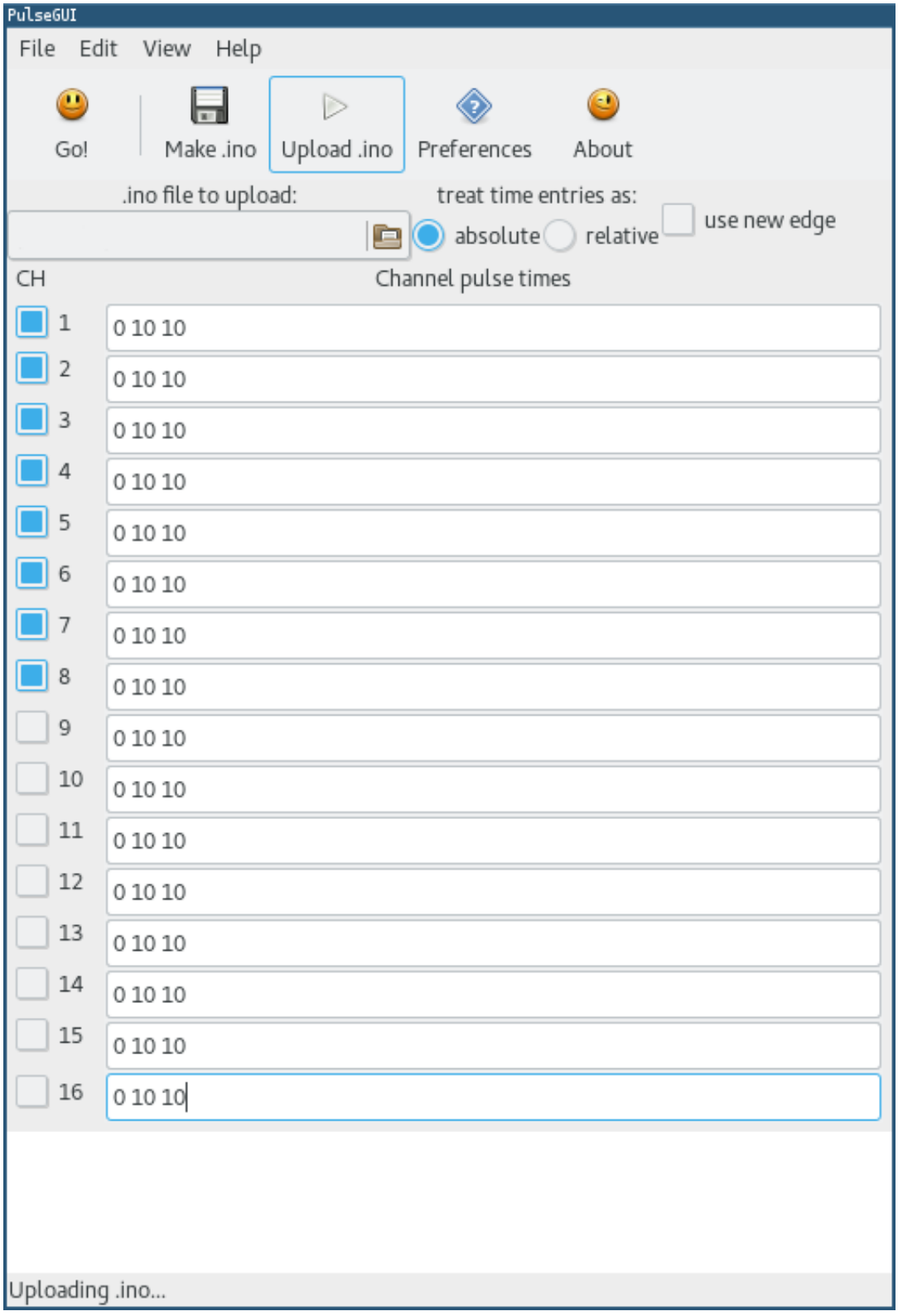


Figure 7: The prototype version of the pulsebox GUI. Each input line represents one channel. Numerical values can be written here to specify delays between channel flips on a given channel. Thus the end user only needs to know the relative temporal displacement between output changes on each of the single channels to specify the entire seqeuence. In this case, the sequence used for a measurement of simultaneous channel switching (described in Section 3.2) is being generated.

3 Testing and measurements

A full characterization of the developed pulsebox needs to be performed. The characteristics I am going to describe and measure are *trigger-to-sequence delay*, the *shortest possible pulse* (achieved both with and without the use of a delay loop between output changes), *simultaneousity* of multiple channels switch during one event, and *sequence jitter*. Additionaly, a *calibration curve* must be obtained for the relation between delay length and a number of passes through the delay loop. This will alow us to convert number of passes to time units and vice-versa. Especially the latter will prove useful in the user interfaces and user-created scripts, where it will allow the user to specify time units instead of a number of passes.

The measurement apparatus for these characteristics and the calibration curve consists of a digital storage oscilloscope (DSO, LeCroy), which was used to acquire output waveforms and perform basic measurements, and a multi-channel time-to-digital converter (TDC, quTAU), which performed precise time measurement of the times of income of rising edges of the pulsebox output. In some measurements and calibration scenarios, a function generator (FG, Tektronix) was used.

3.1 The calibration curve

The calibration curve for the relation between delay length and a number of passes through the delay loop has been obtained using measurements made on a specifically designed set of sequences (see Figure 8), one of which is shown in Listing 9.

```
Set CH1 to HIGH
Delay (for a fixed amount of time)
Set CH1 to LOW
Delay (for a variable amount of time)
Set CH1 to HIGH
Delay (for a fixed amount of time)
Set CH1 to LOW
```

Listing 9: A single sequence from the set of sequences used for the pulsebox calibration.

Two pulses of a constant length are created with a variable delay between them. It is the dependence of the length of this delay on the number of iterations through a delay loop that is measured. Sequences with a large range of delay loop passes are generated, from one pass to one hundred million of passes, then measured with the TDC in the order of rising delay. To span this large range of orders of magnitude, the delays were chosen to form a *pseudo-geometrical series* such as the one shown in (1).

$$\{1;\ 2;\ 5;\ 10;\ 20;\ 50;\ \ldots;\ 10,000,000;\ 20,000,000;\ 50,000,000;\ 100,000,000\} \quad (1)$$

The ideal pulsebox output for these sequences is shown in Figure 8. The TDC measures the times of incoming rising edges. This means that in order to calculate the actual length of the delay between the two pulses, the length of the whole first pulse must be subtracted. Furthermore, each delay event consists of instructions for storing the iteration variable in an internal register of the MCU, which take place before the delay loop itself. The process of storing the value of the iteration variable takes an amount of time which is constant for every delay event. If we subtracted this time interval as well, we would be left with a time interval corresponding to the increment of the delay for which the actual passes through the delay loop are responsible. To achieve the subtraction of both the first pulse

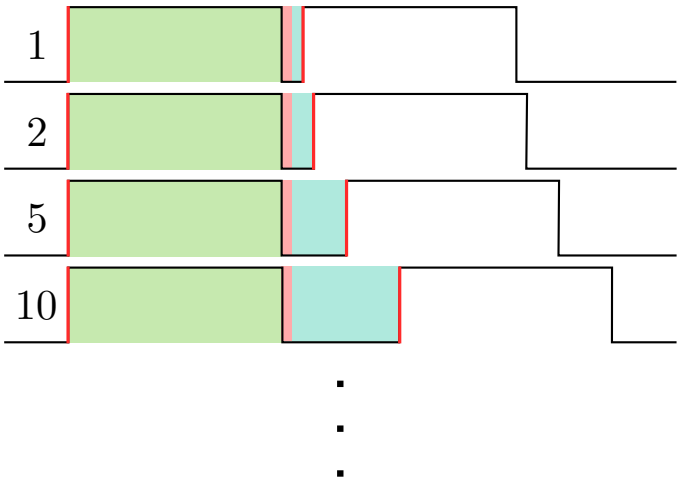


Figure 8: The set of pulsebox sequences used for calibration. The number of iterations through the delay loop between pulses is shown on the left side of each sequence. The TDC measures the time interval between the two rising edges of the output signal (shown red). The pink band represents the delay caused by the instructions used for storing the iteration variable (see Listing 5 for details), which stays constant for each sequence. The cyan band is the part of the delay which is caused by iterating through the delay loop and is responsible for the prolonging of the delay between the two pulses. The green band represents the width of the first pulse, which stays constant throughout the set of the sequences.

length and the initial instruction time, the time interval of the delay measured with the first sequence (with one delay loop pass) is subtracted from the measured delay lengths of all the following sequences. Using this method, we obtain a time interval corresponding to the *prolonging* of the measured delay compared to the first sequence. This prolonging is achieved by an increase of the number of delay loop passes relative to the first sequence. For the example series of loop passes shown above, we would obtain a series of increments of the number of delay passes shown in (2).

$$\{1;\ 4;\ 9;\ 19;\ 49;\ \ldots;\ 9,999,999;\ 19,999,999;\ 49,999,999;\ 99,999,999\} \tag{2}$$

The relation of prolonging of the delay on increase of delay loop passes can be used for calibration of the pulsebox.

The results of calibration are shown in Figure 9. The obtained relation between pulse prolonging and the increase of delay loop passes is almost perfectly linear. A linear relation like the one shown in Figure 9 has only been achieved with inline-assembly-based delay loop solutions. The early attempts at realizing a delay loop using the standard C `for` loop have been discovered to yield much more complicated calibration curves.

The slope of the linear relation is of a great interest to us, as it tells us the time step of pulse prolonging. From the calibration it has been calculated to be 64 ns, which corresponds to a little more than five clock cycles of the MCU. The result is not a multiple of the length of one clock cycle (12 ns). This is because that the result is obtained as an average slope. The slope of the calibration curve changes by a tiny fraction in some areas, which is enough to move the average slope away from the expected value of 60 ns.

One more result can be read from the calibration curve. In the last part of calibration, where the *total* number of delay loop passes was 100,000,000, the temporal prolonging of the delay was almost ten seconds. This means that my method of inline-assembly-based delay loops for the delay event can be used for pulse length setting with a consistent time step over a great range of time intervals, from hundreds of nanoseconds (10^{-7} s) to tens of seconds (10^{1} s). It should be noted that if the linearity was consistent, the maximal time

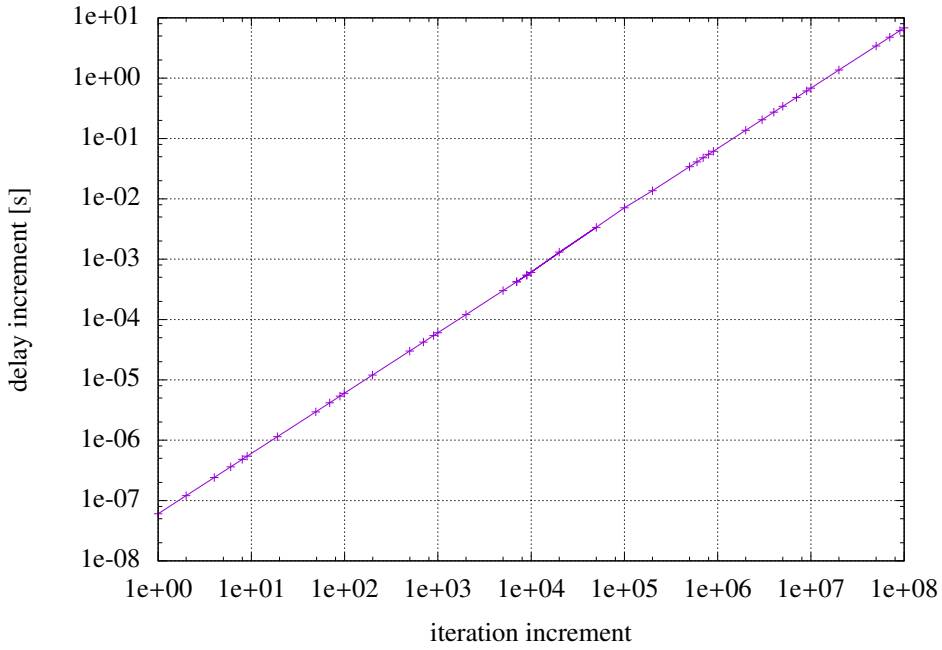


Figure 9: The calibration curve for delay prolonging. The figure shows significantly linear relation between delay prolonging and the increment of delay loop passes. The set of calibration sequences is different from the one corresponding to the example series shown in (1) as it contains an even greater number of sequences to allow more data points to be measured.

interval achievable using a single delay event would be little less than 300 seconds. This limitation is due to the fact that the iteration variable is stored in a 32-bit general-purpose internal register. The highest value of the iteration variable is then 4,294,967,295. The calibration beyond the range of seconds, the method of prolonging delays by *stacking* delay events, and the appropriate discussion of its linearity is, however, beyond the scope of this Thesis, since these time intervals are not relevant to our applications of the pulsebox.

3.2 Other characteristics

As another characteristic of the pulsebox, *sequence jitter* was measured. Principally, when the sequence is repeated over and over again, it might be possible that the length or position of pulses changes. This might occur only once in a great number of repetitions. This jitter might be caused by an MCU interrupt, which delayed the execution of the sequence code. This parameter is crucial, as it greatly influences the repeatibility of measurements and experiments made using the pulsebox. No jitter has, however, been observed. The sequence jitter measured using the DSO was sub-nanosecond. This has also been confirmed during the measurements of other pulsebox characteristics, including the calibration curve, with the TDC.

The shortest pulse was generated using two output changes—`HIGH` followed by `LOW`—on a single channel without inserting a delay event between those output change events. The resulting pulse length was measured by the DSO to be 24 ns. As the MCU of the Arduino Due board we used for the pulsebox prototype runs at the clock frequency of 84 MHz, one clock cycle translates to roughly 12 ns. I propose then that the 24 ns pulse length is the lowest achievable with Arduino Due, as a minimum of two MCU operations is needed to set a channel `HIGH` and then `LOW` again. For our applications, we will always include the delay events between output changes. The shortest pulse achieved by this *delay method* was also measured, this time using a more complicated sequence (shown in Listing 10 and

Figure 10) and a TDC to provide more precise measurement.

```
Set CH1 to HIGH
Delay (one delay loop pass)
Set CH1 to LOW
Delay (one delay loop pass)
Repeat
```

Listing 10: The pulsebox sequence for measuring the length of the shortest pulse achievable using the delay method.

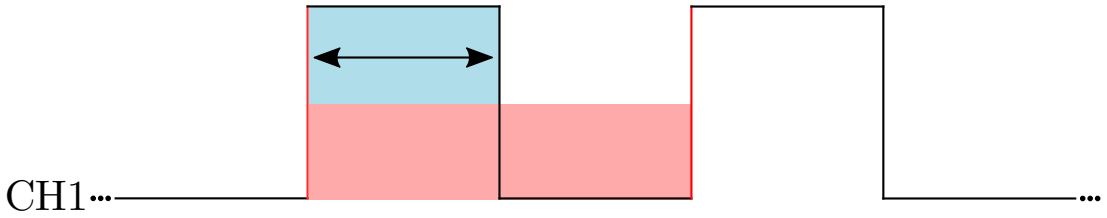


Figure 10: The sequence for measuring the shortest pulse with delay in between. The red band is the time interval measured by the TDC. The blue is the pulse length to be calculated. Ideally, this is a half of the length of the red band.

As the TDC system used in the measurement only reacts to rising edges of the input signal, the sequence takes form of a square signal with a 50 % duty cycle. This way, the time interval between two rising edges of the pulsebox output is measured. This time interval is then divided by two. The result is the pulse length. The comparison of the two techniques of short pulse generation is shown in Figure 11. The result of the measurement of shortest pulse possible using the delay method is 100 ns. The uncertainity in the duration of the resulting pulse (see Figure 12) is around the 12 ns mark, which corresponds roughly to one clock cycle of the pulsebox MCU and has been found to stay approximately constant for longer pulses. Most of our applications do not require pulses of this minimal length, the uncertainity will then be negligible in most cases. It should be noted that the shortest pulse generated using the delay method differs greatly in length from the shortest pulse achieved simply by two consequent output changes. This is due to the fact that when using a delay loop, even when only one pass occurs, a series of instructions is always performed before the body of the loop itself. These instructions are vital to storing the number of iterations of the loop in an internal register of the pulsebox MCU and cannot be, in principle, eliminated.

When we combine the measured length of the shortest pulse achievable using the method of a delay event between two output events and the timing step obtained from the calibration, we can now say that the granularity level of timing for the pulsebox is 100 ns. This is sufficient for most of the experimental and measurent scenarios where the pulsebox will be deployed.

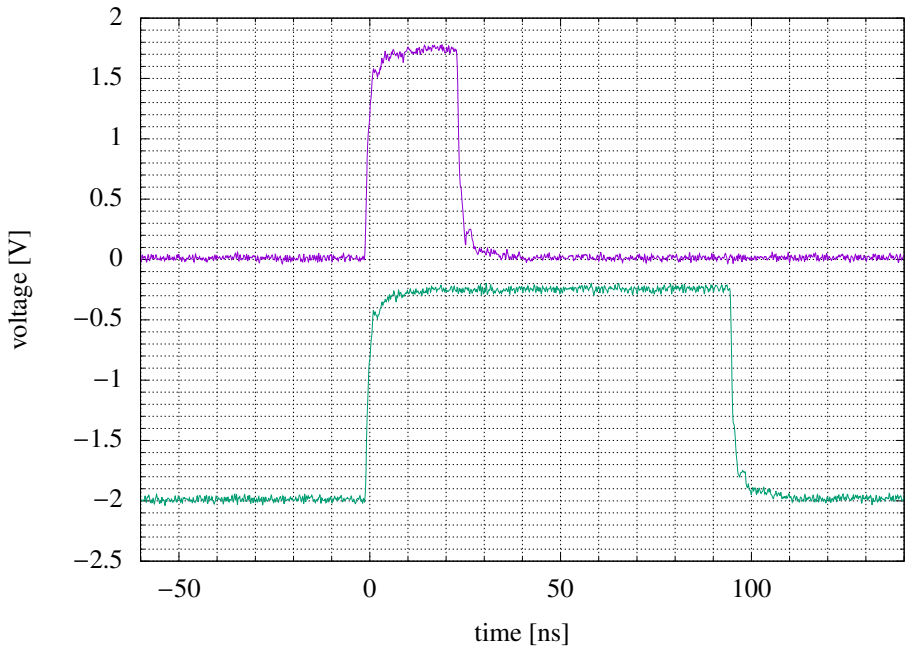


Figure 11: The shortest achievable pulses. The pulse achieved by two consequent output changes cannot, in principle, be shorter than two MCU clock cycles. The pulse generated using one delay loop pass between the output changes is longer, but represents the actual technique for pulse sequence generation used in our pulsebox solution. The green waveform is offset.

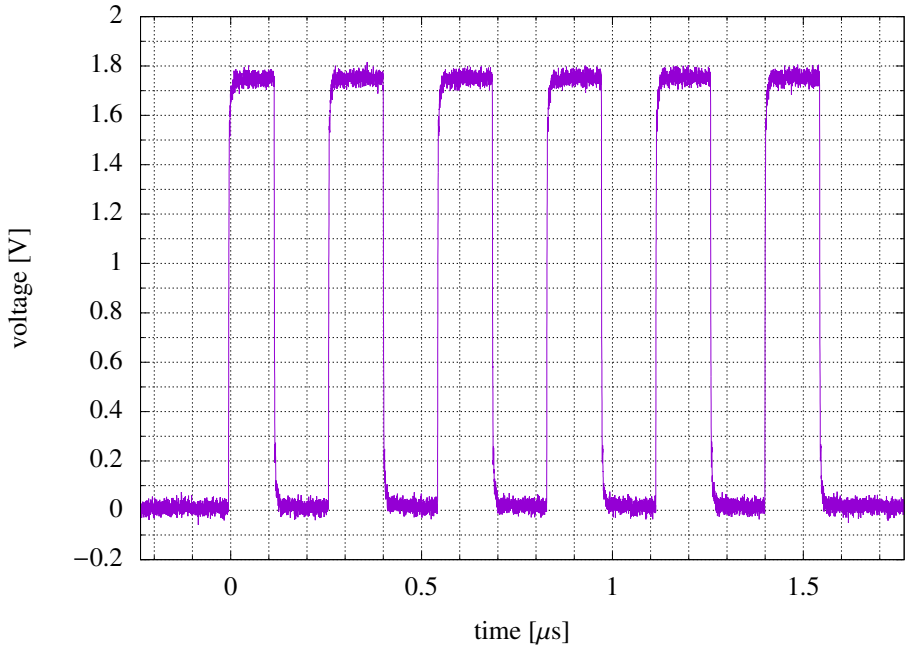


Figure 12: The time uncertainity of pulse generation. Ideally, pulses of equal lengths should appear, but the systematic error of the real pulsebox output is around 12 ns. This is still acceptable for our applications.

To measure the simultaneousity of multiple channels switching during one event, a sequence shown in Listing 11 and Figure 13 was created. Eight channels were switched simultaneously, instead of the full range of 16, as the used TDC has only got eight inputs.

```
Set CH1..CH8 to HIGH
Delay (for an arbitrary amount of time)
Set CH1..CH8 to LOW
```

Listing 11: The sequence for the measurement of simultaneousity of multiple channel switching during one output change event.

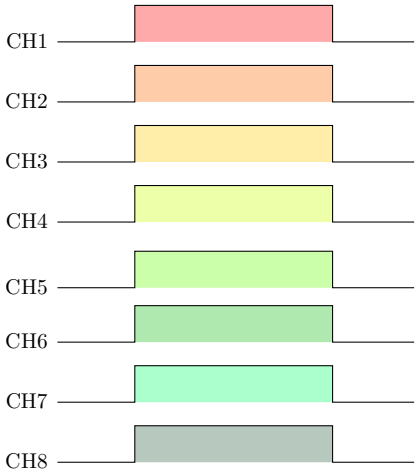


Figure 13: The sequence for measuring simultaneousity of multiple channel switching during one event.

The sequence was repeated many times and the TDC measured the times of the raising edges in the pulsebox output. The times of rising edges of every sequence were compared and the difference between the last and the first incoming edges was computed. The result tells us that when multiple channels are switched during a single event, the time difference between switching of single channels is 1.5(4) ns, which is in accordance with the waveforms captured by the DSO in Figure 14.

The trigger-to-sequence delay tells us how long it takes to the pulsebox to react to the external trigger and launch its sequence. To measure this, a sequence was designed with one channel set to `HIGH` as soon as the sequence started. The trigger signal was a short pulse generated using the FG with period set to a time long enough to allow the sequence of the pulsebox to finish before triggering it again. This trigger signal was generated using both of our FG's output channels running in synchronized mode to allow us to use one FG channel to trigger the pulsebox and the other channel for measurement of the trigger-to-sequence delay using the TDC. The single-channel output of the pulsebox was also fed to the TDC. The delay was computed by subtracting the time of the incoming raising edge of the trigger signal from the rising edge of the pulsebox sequence. The trigger-to-sequence delay has been measured (see Figure 15) to be under 3 μs. Furthermore, it has been found that this delay can differ depending on the actual sequence that is generated using the pulsebox. The jitter of the delay is around 10 ns, which is negligible compared to the length of the delay. The length of the trigger-to-sequence delay is of no concern, because the main applications of the triggered mode of pulsebox are synchronization of the sequence with a DSO or with a background noise induced by electrical outlets, where this length of a trigger-to-sequence delay is negligible.

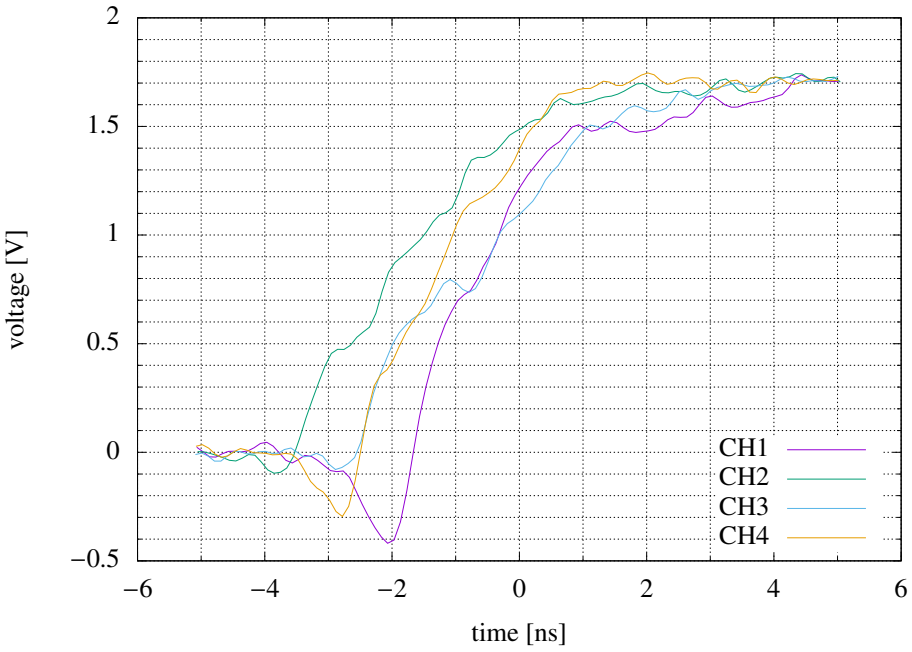


Figure 14: Simultaneousity of multiple channel switching during one event. The waveform is shown as obtained using the LeCroy oscilloscope. This is why only four channels are shown instead of eight. This figure is for illustratory purposes only.

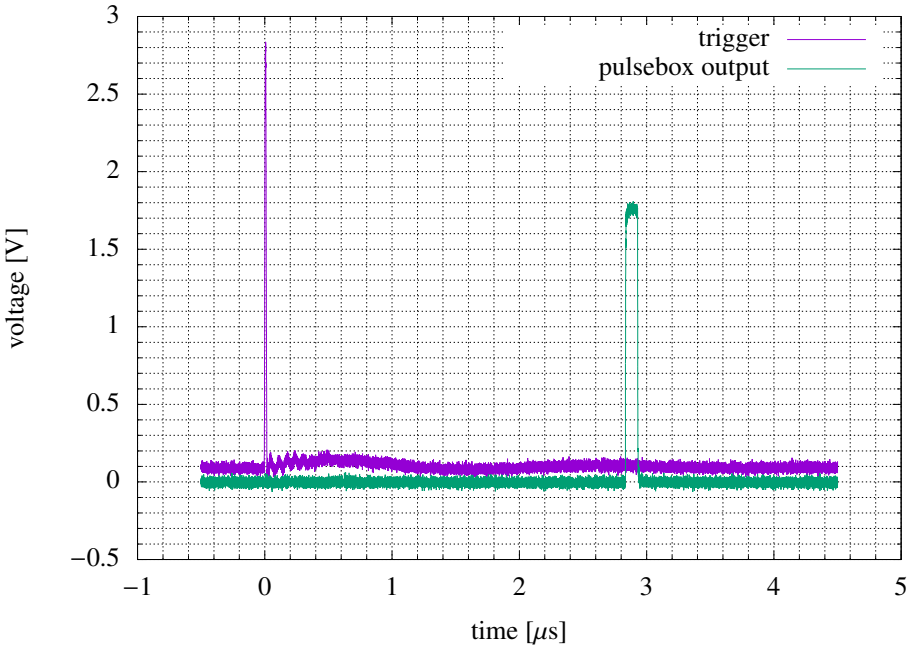


Figure 15: Trigger-to-sequence delay.

The more events a pulsebox sequence contains, the more code blocks are used in the source code `.ino` file. This should mean that the size of the compiled program code for the MCU should rise as well. This has been confirmed. The amount of memory that the MCU used in the Arduino Due developement board reserves for the code is 524,288 bytes. It has been calculated that with each added code consisting of a delay event and an output change event (a *delay-output change pair*) there is a 24 byte increase in the size of the

compiled code. Keeping in mind that the code representing even an empty sequence still has some size, it has been estimated that the memory capacity of the Arduino Due's MCU is enough to allow a total of 20,000 of these delay-output change pairs in our sequences. We will call this number the *event capacity* of the pulsebox.

4 Application

The pulsebox was deployed in a stopped-light experiment, which relies on the phenomenon of electromagnetically induced transparency (EIT) in a cavity with rubidium (Rb) vapors. Under normal conditions, an optical *signal beam* will not pass the cavity as it is opaque. However, when applying a strong optical *control beam*, with its frequency near the atomic resonance of the vapors, to the cavity, its transparence spikes rapidly and allows for the signal beam to pass through the cavity. By switching the signal and control beam on and off in a correct manner, and with precise timing, it is possible to achieve slow-light or even stopped-light behaviour of the signal beam. More about the phenomenon of EIT and its applications can be found in [15]. One possible application of the slow-light and stopped-light conditions induced by the EIT is a quantum memory which stores a photon with information encoded into its quantum state.

The stopped-light phenomenon has been observed (see Figure 16) in an EIT experiment built by Lukáš Slodička and Petr Obšil. The electronical pulses generated by the pulsebox were converted to optical pulses using an AOM. This shows that the MCU-powered pulsebox can be readily used in experimental settings.

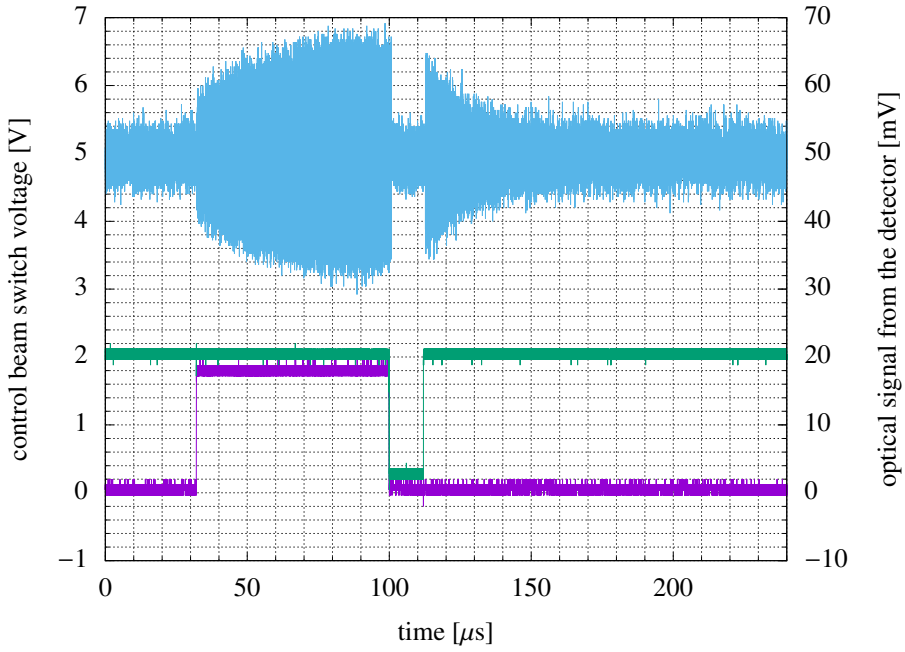


Figure 16: The results of the stopped-light experiment. The voltage of the signal switch (purple) is pulsed, creating a pulse of the signal beam. At the end of the pulse, a pulse of the voltage on the control beam switch (green) occurs, toggling reflectivity of the vapor cavity. The blue waveform is the signal from the optical detector at the end of the cavity. It shows the signal pulse, which is stopped for roughly 10 μs, then released again, without any input signal for the signal beam switch. Note that the purple and green waveforms are ofset by a little amount and the blue waveform is offset.

5 Conclusion and outlook

A concept of an MCU-powered multi-channel digital event sequencer prototype has been created. It relies on an event queue, in which events—digital channel output changes and delays—are stored. For each event there is a corresponding piece of C code. When these so called code blocks are put together, a source code file is created, which is then uploaded to the MCU of the pulsebox.

A prototype pulsebox based on this concept has been built and a software interface has been implemented to allow for its remote programming. The parameters of the device, summarized in Table 3, meet our requirements.

calibration curve slope (see Figure 9)	64 ns
sequence jitter	sub-nanosecond
shortest pulse (delay method)	100 ns
granularity	100 ns
simultaneousity of multi-channel switching	1.5 ns
trigger-to-sequence delay	3 μs
trigger-to-sequence delay jitter	10 ns
event capacity	20,000 of delay-output change pairs

Table 3: A summary of the parameters of the pulsebox.

The functionality and usability of the pulsebox has been demonstrated in a stopped-light experiment based on the phenomenon of electromagnetically induced transparency EIT.

There are, however, ways to extend the functionality of the prototype to allow it to be used in a wider array of experimental applications. One of the ways to improve the pulsebox is low-impedance termination of the pulsebox outputs. This can be achieved by using a operational amplifier (op-amp) as a buffer for the output pins of the Arduino board. Output buffering would also allow us the *doubling of channels*. It is often desirable to split the output signal of a pulsebox channel into two to, for example, connect it to an electronical device used in an experiment and also to an oscilloscope. With the current prototype, attempting this would reduce the voltage of the output by a factor of two. Two pulsebox channels with the same output have to be used instead. An op-amp could also be used as an amplifier. This would allow us to perform *logical level conversion*. The digital pins of the MCU on the Arduino Due board output 0 V in their `LOW` state and 3.3 V in their `HIGH` state. This is still compatible with the transistor-transistor logic (TTL) standard, as it considers any voltage above 2.7 V to be the logical `true`, or `HIGH` in our terms. It would be sensible, however, to convert the 3.3 V logical level for `HIGH` of the Arduino to the 5 V of TTL. In principle, level conversion to other standards of digital logic would be possible as well.

Furthemore, the use of the pulsebox for generation of an analog signal, RF or optical, is desirable. This could be achieved in many ways. In conjunction with a VGA amplitude modulation of optical pulses (*pulse shaping*) could be achieved and using a DDS an RF signal could be generated to drive EOMs and AOMs, resulting in phase and frequency modulation of an optical signal. These applications must be further tested.

Finally, the measurements of the pulsebox characteristics and its calibration curve could be made more simple and precise by using a TDC that registers not only the times of the incoming rising edges, but also *falling* edges. The set of calibration sequences would then consist only of single pulses of varying lengths, which could be measured more accurately. Alternatively, the calibration sequences can be run on two channels in parallel. The second channel would output the inverted sequence, changing falling edges to rising

and allowing us more precise measurements even with the TDC sensitive only to rising edges of the input signal.

Another area of improvement is adaptive sequence correction. When high timing precision of a sequence is required, the output of the pulsebox could be measured by the TDC system and compared to what the specified sequence should ideally look like. If, for example, a single pulse was wider than desired, possibly due to an imperfect calibration, this could be corrected by decreasing the number of passes through the corresponding delay loop. The corrected sequence would then be uploaded to the pulsebox and measured again. This process could, in principle, be repeated to achieve even better results. To correct the sequence in this manner would be much faster than performing the full calibration of the pulsebox with delays spanning many orders of magnitude. It is however important to note that it is only possible to correct the timing discrepancies larger than the pulsebox granularity.

References

[1] M. R. Dietrich and B. B. Blinov. Use of a microcontroller for fast feedback control of a fiber laser. *arXiv:0905.2484v1*, 2009.

[2] D. Nino, H. Wang, and J. N. Milstein. Rapid feedback control and stabilization of an optical tweezers with a budget microcontroller. *Eur. J. Phys.*, 35:055009, 2014.

[3] D. J. Saunders, J. H. D. Munns, T. F. M. Champion, C. Qiu, and K. T. Kaczmarek. Cavity-enhanced room-temperature broadband Raman memory. *Phys. Rev. Lett.*, 116:090501, 2016.

[4] K. C. Cox, G. P. Greve, J. M. Weiner, and J. K. Thompson. Deterministic squeezed states with collective measurements and feedback. *Phys. Rev. Lett.*, 116:093602, 2016.

[5] K. Huang, H. Le Jeannic, J. Ruaudel, O. Morin, and J. Laurat. Microcontroller-based locking in optics experiments. *Rev. Sci. Instrum.*, 85:123112, 2014.

[6] S. Kasas, L. Alonso, P. Jacquet, J. Adamcik, C. Haeberli, and G. Dietler. Microcontroller-driven fluid-injection system for atomic force microscopy. *Rev. Sci. Instrum.*, 81:013704, 2010.

[7] V. Bocci, Giacomo Chiodi, Francesco Iacoangeli, Massimo Nuccetelli, and Luigi Recchia. The ArduSiPM a compact transportable software/hardware data acquisition system for SiPM detector. In *2014 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC)*, 2014.

[8] N. C. Ryder. Microcontroller based data acquisition system for silicon photomultiplier detectors. *J. Instrum.*, 8:C02019, 2013.

[9] B. Kristinsson. Ardrand: The Arduino as a hardware random-number generator. *arXiv:1212.3777v1*, 2011.

[10] M Sadgrove. Microcontroller interrupts for flexible control of time critical tasks in experiments with laser cooled atoms. *arXiv:1104.0064v1*, 2011.

[11] E. E. Eyler. A single-chip event sequencer and related microcontroller instrumentation for atomic physics research. *Rev. Sci. Instrum.*, 82:013105, 2011.

[12] P. E. Gaskell, J. J. Thorn, S. Alba, D., and A. Steck. An open-source, extensible system for laboratory timing and control. *Rev. Sci. Instrum.*, 80:115103, 2009.

[13] T. Pruttivarasin and H. Katori. Compact FPGA-based pulse-sequencer and radio-frequency generator for experiments with trapped atoms. *Rev. Sci. Instrum.*, 86:115106, 2015.

[14] L. Sun, J. J. Savoy, and K. Warncke. Design and implementation of and FPGA-based timing pulse programmer for pulsed-electron paramagnetic resonance applications. *Concepts Magn. Reson.*, 43:100, 2013.

[15] I. Novikova, R. L. Walsworth, and Y. Xiao. Electromagnetically induced transparency-based slow and stored light in warm atoms. *Laser Photonics Rev.*, 5:333, 2012.