Seamless Integration of Hardware Components: Real-Time Programming for Streamlined Functionality in Nociceptive System Analysis

Adam Andreas Tønsberg - s2340283

Contents

Abstract

Accurate assessment of biological systems like the nociceptive system is essential for studying the sensory perception of pain. This paper introduces a refined approach to pattern generation for a nociceptive stimulator. Contemporary research has shown that electrical stimulation in the form of short pulses can activate the $A\delta-$ and $C-$fibers that are used in studying nociceptive behavior. By configuring the pulses and their frequency, and reading the evoked brain response, biomedical researchers can better model the nociceptive system[1]. However, accurate timing of the stimulation pulses in a particular pattern is of primary importance.

By setting up a timer interrupt for the pattern generation directly within the stimulator's firmware, the system achieves exceptional timing precision. This shift simplifies the setup, reduces the need for specialized knowledge such as calibration constants, and makes it more accessible beyond lab settings. The resulting compact system can be applied to studying nociceptive behavior and pain perception.

1 Introduction

Individuals suffering from chronic pain form a large demographic in healthcare consumption, and so gaining an understanding of how chronic pain is developing in patients - its early diagnosis - can bring about a greater degree of control in early treatment of these symptoms, increasing the chances of a successful treatment. The goal, for researchers, is to develop methods to understand the nociceptive processes - how the human body reacts to stimulation.

The *Ambustim* is a device that was developed to generate electrical stimulation in the form of rectangular pulses. By stimulating human subjects with a certain pulse characteristics (like width and amplitude), the detection threshold of the nociceptive nerve is determined. Studying the detection threshold on various subjects is one of the approaches taken by biomedical researchers to expand the understanding of pain perception.

Unfortunately, the current implementation of the system comes with several hurdles that need to be overcome in order to expand the research[2]. One of the issues is that the duration of these experiments is impractical, up to 2 hours are required in attaining meaningful results [3]. Another major issue is that the setup requires a lab setting and is unfeasible for untrained physicians. This is because of the high complexity and limitations of the current design.

Contributing to the complexity of the system, when researchers wanted to understand the relationship between issuing stimuli in certain patterns and evoked brain responses, a new feature was introduced for this purpose. This new feature of modulated stimulation pulses has introduced additional impracticalities to the system. First, in the form of additional hardware, with additional wired interconnections to the pattern generating device. Second, new limitations in-terms of what the system is capable of reliably configuring to. Third, a separate application for configuring for the new hardware. Originally, a single application is used to communicate with the stimulator, wirelessly, via Bluetooth. Now, a second application interfaces with the pattern generator through a wired USB connection.

This paper explores alternative design choices for improved functionality integration, for reducing system complexity and increasing accessibility. Section 2 is a comprehensive analysis of the current system and relevant literature to establish the requirements for an improved design. Informed by these requirements, Section 3 covers the development of a new design. In Section 4, the testing process is covered with the outcomes of the new design. Then, Section 5 discusses the outcomes of the new design compared to its previous implementation, providing recommendations for further improvements. Finally, Section 6 concludes the paper.

2 Requirements

2.1 General background

Electrical stimulation can be used to activate the nociceptive system in a repeatable way. This means that reactions can be consistently observed, making it possible to detect patterns using Electroencephalogram (EEGs). The use of EEGs can offer a deeper insight into nociceptive behavior[4]. The system works by utilizing electrodes that lightly touch the skin to deliver the stimuli.

Employing an EEG cap to monitor brain reactions to these stimuli can further enhance understanding of the nociceptive process. Responses to these stimuli are also indicated by users through a button release.

Studying faulty aspects in the nociceptive system of chronic pain patients is complicated by the nonlinear relation between the stimulus and response. So, in addition to characterizing the stimulus-response pairs by using a single transient stimulus, biomedical researchers are using an alternative technique to characterize stimulus-evoked brain activity; by measuring Steady State Evoked Potentials (SSEPs). SSEPs rely on sustained activation of brain areas by a continuous sensory stimulus. By modulating the intensity of this stimulus with one or multiple frequencies, evoked brain activity can be observed in the electroencephalogram (EEG) at these frequencies and their harmonics. [5]

The **Ambustim** is a device that performs the task of generating the customized stimulation pulses. To achieve the continuous sensory stimulus for SSEPs, the system is designed to receive patterns from a peripheral device in the form of (trigger) pulses. This pattern generating device is called the **trigger box**. Upon receiving a trigger a stimulus is given, this way the stimulus pattern will follow the pattern at which the trigger pulses are being received. The setup is illustrated in Figure 1.

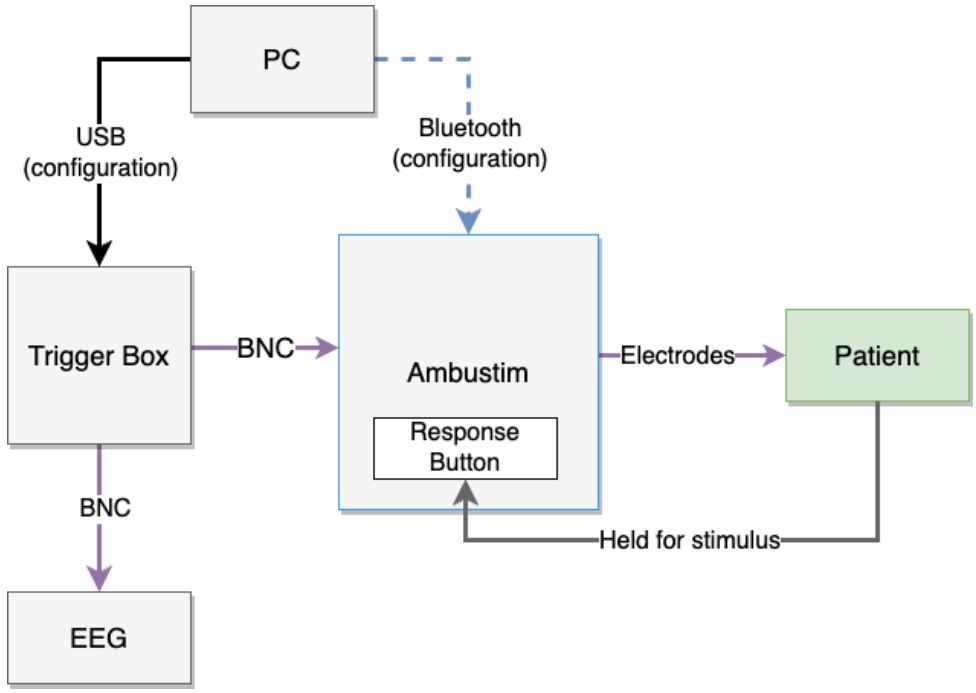

Figure 1: Interconnections of the current system

Figure 1 shows the current setup used. From here the functionality of the `trigger box` is made clearer: the `trigger box` sends the trigger pattern to the *Ambustim*, and the first trigger in a trigger pattern is also sent to the EEG device to signal the start of an evoked potential response.

2.2 Ambustim

The *Ambustim* is run by an ATmega162 microcontroller unit that controls the subsystems of the device. Most notably is the Digital-to-Analog Converter (DAC) that is utilized for generating high voltage pulses meant for stimulating the nociceptive nerves. The embedded code includes real-time programming which allows for precise control over the microcontroller's operations. [6]

2.2.1 Real time programming

Real-time embedded systems programming involves interacting directly with a microcontroller's hardware registers, providing precise control and efficient use of the microcontroller's resources.

Specifically, this means directly configuring the hardware peripherals, such as timers, communication interfaces, etc.

In contrast, the Arduino framework abstracts away this direct control, which is beneficial for a quick implementation of a device's functionality. However, it often limits the optimization that can be obtained via directly accessing registers.

The ATmega162 is part of the AVR microcontroller family, which provides a mechanism for interrupting the main loop. This feature allows the interruption of ongoing tasks to handle designated events through Interrupt Service Routines (ISRs). Features like the interrupt mechanism are critical for managing time-sensitive operations in real-time programming scenarios.

Hardware timers can be set up to automatically count up or down. When the timer reaches a certain value, an interrupt flag is set by the hardware. The CPU, while executing its main program, regularly checks these flags. When a flag has been set, it pauses the main program, saves the current state, and executes the associated ISR. Once the ISR is completed, the CPU restores the previous state and resumes execution of the main program.

2.2.2 Main loop

The embedded code has 4 Interrupt Service Routines (ISRs) being used. (1) **USART interrupt**: listens for incoming commands via the USART line, which connects to the Bluetooth module. These commands are stored in a string array. (2&3) **Timer3 comparator A and B**: these ISRs handle timing events associated with the different phases of stimulation pulses. (4) **Timer1 comparator B**: used in setting the `maximum response time`, a parameter that functions as a timeout for the experiment, ensuring that the stimulation sequence is terminated within a specified period if a response is not recorded via button release.

```
SIGNAL(TIMER3_COMPA_vect){
    // Handles device states STIM_PHASE_4, STIM_PHASE_5, STIM_PHASE_6
    // Defaults to IDLE state
}

SIGNAL(TIMER3_COMPB_vect){
    // Handles device states STIM_PHASE_1, STIM_PHASE_2, STIM_PHASE_3
    // Defaults to IDLE state
}

SIGNAL(USART1_RXC_vect){
    // receives commands from USART line
}

SIGNAL(TIMER1_COMPB_vect){
    // Stops timer1
}
```

The main loop will check for any new commands to parse, and configure the parameters of the stimulation pulses. If a trigger is received and parameters for the pulse train are used to configure the stimulation pulses, the `start_pulse` function is called, initializing Timer3 and the first phase of the stimulation pulse begins (positive pulse, denoted as `STIM_PHASE_1`). In context of the illustration in Figure 2, this is the `STIMULATION` state. During this time, the main loop continues to scan for new commands, but the ongoing pulse train is unaffected by these inputs.

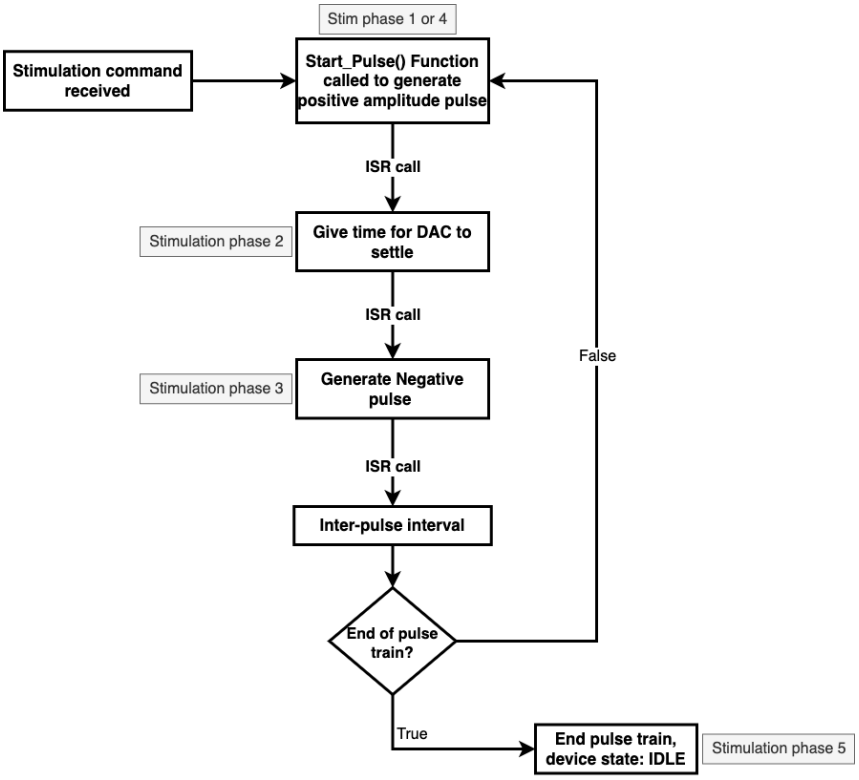

Figure 2: System timers flow diagram

2.2.3 Timers

The ATmega162 provides several options for pre-scaling, including 8, 64, 256, etc. Prescalers are mechanisms used to slow down the timer of the microcontroller by dividing the clock speed by the prescaler factor. For example, a prescaler of 256 divides the clock speed by 256, slowing down the timer by this factor. The choice of prescaler must ensure that the timer does not overflow during the required period; in other words, it must be sufficiently long so that no overflow occurs. The current design has Timer3 (a 16-bit timer) set to a prescaler of 256, which, given a time unit of 34.7 microseconds (=1/7.3728 MHz), results in $(2^{16} - 1)/7.3728$ MHz, or 2.3 seconds, before the timer overflows.

As illustrated in Figure 3, Timer 3 alternates between its comparators, i.e., when Comparator 3A interrupt is enabled, Comparator 3B is disabled, and vice versa. Comparator 3A checks whether the pulse sequence is terminated or has more in its sequence. This alternating mechanism is a useful way of organizing the comparators for different tasks; in practice, ISRs (Interrupt Service Routines) should be kept as short as possible, executing only a few operations within their scope.

Using the ATmega162 datasheet[6], the register variables can be accessed and programmed directly in software, meaning that specific bits in the register can be toggled to enable certain modes of operation. When dealing with timers, two of several key registers include: the Timer Interrupt Mask Register (TIMSK) and the Output Compare Register (OCR). The Timer Interrupt Mask Register includes the Output Compare Interrupt Enable (OCIE) bit, which governs the microcontroller's responsiveness to timer interrupts. On the other hand, the OCR sets the time interval before triggering the next interrupt.

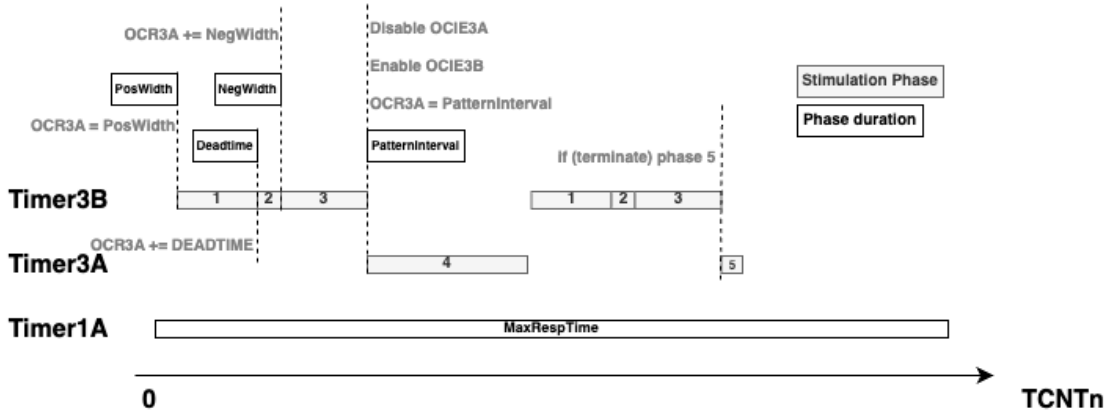

Figure 3: Stimulation phases, showing the timer relevant register calls. TCNTn (Timer/Counter Register) refers to the different timers where n is the timer number

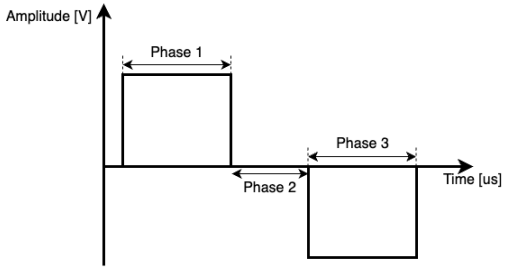

Figure 4: Illustration of the pulse widths and amplitudes depend on the parameters given in the commands.

2.2.4 Stimulation phasing

The device states are organized using an enumeration, which defines the different stimulation phases and an idle state, as shown below:

```
typedef enum {
    // Stimulation phases
    STIM_PHASE_1, // Positive pulse
    STIM_PHASE_2, // DeadTime
    STIM_PHASE_3, // Negative pulse
    STIM_PHASE_4, // Interpulse interval
    STIM_PHASE_5, // Terminate S command
    STIM_PHASE_6, // Control Q command (DEPRECATED)
    // Not stimulating
    IDLE
} DeviceState;
```

To reiterate, once `start_pulse` is called, the OCR is set to the positive pulse width, and the interrupt for timer Comparator 3B is enabled. At the first interrupt (end of the positive pulse width), the second phase of the pulse begins, denoted as `STIM_PHASE_2`. This phase includes a period of 'dead-time' during which the OCR is incremented by the `DEADTIME`, allowing the DAC to settle - to reach steady state before the next phase. This is denoted as `STIM_PHASE_2`. After this discharge time, the 3rd phase commences i.e. the negative pulse (`STIM_PHASE_3`). Both phase 1 and 3 check to see whether the parameters are configured to have positive or negative amplitudes greater than 0, otherwise the DAC remains closed.

Finally, at the end of phase 3 the program checks if the pulse train is complete, if so the termination phase commences (declared as phase 5 in the program), otherwise the program is running the inter-pulse interval phase before calling start pulse and transitioning back to phase 1. Note, unless the program is put in the termination phase (phase 5) a new pulse train cannot commence, i.e. start_pulse can not be called from the main loop and will only be called inside of phase 4. This cycle is illustrated in Figure 3.

2.2.5 Stimulation Pulse Parameterization

The commands used to configure the *Ambustim* are communicated using protocols that start with a letter, and uses the comma as a delimiter. For code readability, the commands are compiled in an enumeration as follows,

```
typedef enum {
    FEATURE  = 'F', // Features query (Properties)
    VERSION = 'V', // Version query
    POS_AMP = 'A', // Set Positive Pulse Amplitude
    NEG_AMP = 'a', // Set Negative Pulse Amplitude
    POS_WID = 'W', // Set Positive Pulse Width
    NEG_WID = 'w', // Set Negative Pulse Width
    INTERPULSE_INTERVAL = 'I', // Set interpulse intervals for stimulus
   pattern
    CHANNEL_ORDER = 'P', // Set channel order for stimulus pattern
    CHANNEL_ENABLER = 'C', // Enable or disable channels
    POWER_ENABLER = 'M', // Set Power On/Off
    CHECK_RESPONSE = 'R', // check response button(, IOK) and TRIG
    STIM_MODE = 'S', // Set stimulation mode (NumTriggers, NumPatterns,
   MaxRespTime)
} Commands;
```

2.2.6 Parameter Validation

In addition to checking if the required number of parameters for each command is received, the embedded software does a few checks to make sure the hardware is capable of producing the desired outcome. One such check is done in the pulse width parameter. Each pulse parameter is ensured to be atleast $105\mu s$ wide,

```
if (Par[ParNo] < PW_MIN){
    Par[1] = PW_MIN;
}
```

However, there is no validation of an upper bound in the current implementation. So the maximum will simply be according to the number of bits in the type, in this case 16 bits so $2^{16} - 1$. For example, sending a value larger that what can be represented by 65,535 will result in the value being rewritten as the modulo of 65,535.

Additionally, the DAC used in the device for generating variable voltage amplitudes, is 14 bits. This means that an upper bound must be set to make sure the requested amplitude is within the DAC's capability,

```
Par[ParNo] = Par[ParNo] & (0x0fff);
```

2.3 Trigger Box

The software implementation of the trigger box was initially developed as a temporary solution to allow the exploration of different trigger patterns on nociceptive behavior. Within the trigger box is a simple program that runs on an Arduino Uno board, utilizing delay loops to generate the pattern of triggers sent to the *Ambustim*. This raises a few challenges that motivated a few of the design choices in the current embedded code which will be explored in the next subsection.

To reiterate, the triggers are meant to time the occurrence of stimulation pulses. For example, for a single stimulation pulse, and `num_triggers` = 3, then each pulse will occur when the next three triggers received for the trigger box.

In order to study the system, the code was made more modular, and validation checks were added. These adjustments helped in understanding the underlying mechanisms of how triggers are handled in the Ambustim's main loop and how the trigger box system interacts with it. A flow diagram was constructed to further illustrate these interactions (check Appendix section A.1).

The next step is in analyzing how triggering is processed by the Ambustim. The Ambustim periodically checks the status of the pin connecting it to the trigger box, if HIGH then a trigger has been received. However, a trigger can only be acknowledged if the 'S' command (case `STIM_MODE`)

has been processed by the Ambustim, otherwise the trigger is ignored. This is to make sure stimulation pulses have been configured.

As illustrated in Figure 1, the system relies on wired communication with the trigger box via USB, communicating the following commands for the different trigger modes:

2.3.1 Trigger mode 1: Regular Trigger

- Command: **X** Number **Y**

    1. Number = trigger number to be sent to EEG amplifier, must be between 0 and 255.

The `Regular Trigger` is responsible for initiating a stimulation pulse. Upon activation, it sends a trigger signal to the EEG device, thereby tagging the corresponding response with a numerical identifier. The trigger pulses are 2ms wide, a duration that has been determined to be sufficient for clear detection by the EEG device port, which samples at 1024 Hz.

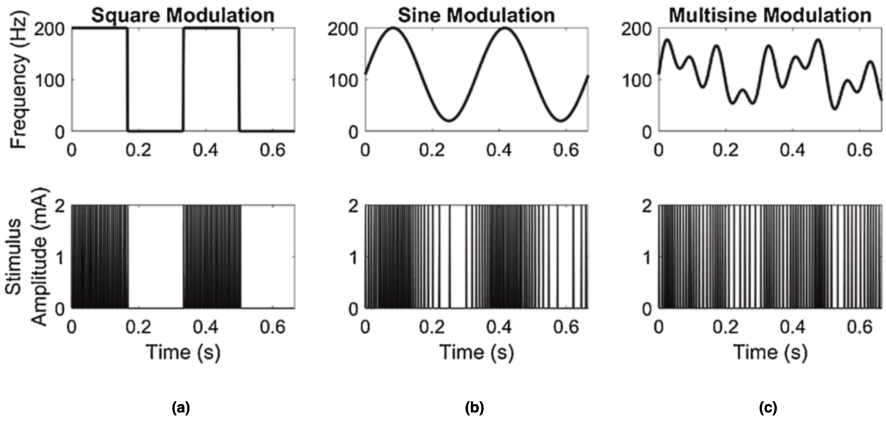

Figure 5: Illustration of modulated trigger pulses. Multisine is based on several frequency components. [5]

2.3.2 Trigger mode 2: Square wave modulated pulse sequence

- Command: **A** Duration, 0, Duty Cycle, Frequency, 0, 0, Interval calibration, Period calibration, 0 **B**

    1. Sequence Length = sequence duration [ms].
    2. Duty Cycle = duty cycle of square wave modulation in [integer].
    3. Frequency = frequency of square wave modulation [Hz].
    4. Interval calibration = calibration constant to correct interval between two pulses [us].
    5. Period calibration = calibration constant to correct duration of the period [us].

The `Square Wave Modulation` pattern is meant to send 2ms triggers in sequence with an inter-pulse interval of 5ms. The duty cycle and frequency of these pulses are communicated, as well as the duration of the pattern. For example a duty cycle of 50% at 3Hz for some sequence length will produce pulse chunks as illustrated in Figure 5a. The calibration parameters are covered in the following subsection.

2.3.3 Trigger mode 3: Multisine wave modulated pulse sequence

- Command: **C** Duration, Frequency Offset, Amplitude1, Frequency1, Amplitude2, Frequency2, Amplitude3, Frequency3, Interval Calibration **D**

    1. Duration = sequence duration [ms].
    2. Frequency Offset = frequency offset of multisine [Hz].

3. AMP1,3 = frequency amplitude first sinusoid[Hz].
4. FREQ1,3 = modulation frequency first sinusoid [Hz].
5. Interval Calibration (PHI) = phase shift of second (+PHI/12) and third (-PHI/12) frequency component.

This mode implements a multisine modulation, which is a combination of three sinusoidal frequency components, each with its own amplitude. The 'Frequency Offset' parameter is used to ensure that all resulting modulation frequencies are positive.

The 'PHI' parameter offers phase shifting capabilities to the frequency components. Achieving the maximum bandwidth per frequency component. When the frequency components are in phase with one another, their amplitudes combine, leading to a narrower effective bandwidth for each component.

Inspection into the underlying real time implementation used for pulse frequency modulation in the arduino revealed 2 requirements: (1) start with (emitting) a pulse at t=0, (2) then using the computed instantaneous period values for subsequent pulses.

2.3.4 Timing inaccuracies

The Arduino code running on the trigger box will calculate the occurrence of the pulses in real time which introduces a number of challenges. For example, the use of software delay loops gives about 15% error in the interpulse intervals when the calibration constants are set to 0. These errors were found using the command parameters for a 3 Hz square wave modulating pattern, at 50% duty cycle for 1 second.

Essentially, the pattern for the square wave implementation is meant to be 2ms pulses separated by a 5ms interpulse interval for the duration of the `on_period`. This means the frequency of the pulses is intended to be about 142.86 Hz, however, with an interpulse interval that is less than 5ms the actual frequency is greater; 200 Hz in this example (check Figure 12):

```
// Square Wave command with suggested calibration constants
A1000, 0, 50, 3, 0, 0, 2030, 702B
```

Consequently, this design choice of correcting for the inaccuracy of the interpulse interval raised an impracticality in the implementation in the form of `calibration constants`, where the trigger intervals were tested to see if they were timed correctly and any inaccuracy would introduce those calibration constants to subtract from the while loop iterations. This was intended as a work-around to reduce the error but had to be empirically determined.

```
// Wait until off period has elapsed.
p_wait = micros();
while(1){
p_current = micros();
if( abs(p_current - p_wait) > offPeriod - periodCalibration ) { break; }
}
```

Time sensitive events can run asynchronously, so that the processor can continue with other tasks. However, this requires a more sophisticated approach than delay loops, that involve interrupting the main loop to execute time sensitive events.

2.4 Design Requirements

(1) Timing Precision: The system must eliminate software timing loops, especially for tasks sensitive to microsecond-level precision. Calibrated constants used for timing are not ideal for this system due to their inherent variability.

(2) Compactness: For practical operation, the device's design should be as compact as possible to increase portability.

(3) Parameter Validation: As new functionalities are added to the system, appropriate validation checks must be setup. This is to ensure expected behavior under operating conditions.

(4) Memory Efficiency: Considering the constraints of the system's memory, it's crucial to optimize memory usage, ensuring that the system operates reliably without exhausting available resources. The aim is to remove obsolete logic, unused variables, and enhance overall readability.

(5) Code Modularity and Extensibility: The codebase should be modular to ensure ease of maintenance and facilitate future development. This involves structuring the software in a way that allows for scalability and ease of integration for new features.

3 Development

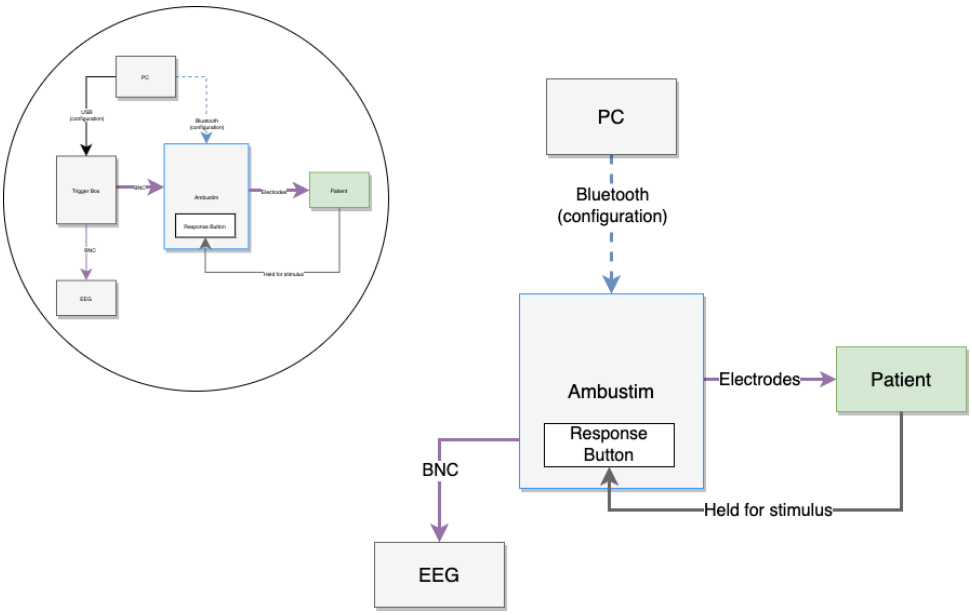

Figure 6: Redesign of the system

Following the design requirements in the previous section, or implications from Figure 6, several solutions are possible. The simplest of these solutions is to replace the trigger box by integrating a second microcontroller (smaller and more powerful than the Arduino Uno) inside the Ambustim's casing. This microcontroller would generate and emit triggers in the same manner as the original trigger box. While this approach has the benefit of simplicity and the ability to handle floating-point calculations more efficiently, it still suffers from timing inaccuracies and requires empirically determined calibration constants. Due to these drawbacks, a more sophisticated software-driven solution for trigger generation was chosen.

The approach for precise timing involves direct register programming to manage delays. Unlike solutions that rely on software delay loops for timing, this approach utilizes the hardware peripherals of the microcontroller. By directly interacting with the timer hardware within the microcontroller, delay periods can be determined more accurately. Timer peripherals can communicate directly with the CPU through interrupts, enabling precisely timed execution of specific functions.

The process of checking whether a pin is high or low can be executed by using a flag that is set for a pulse length of $2ms$, and the following interpulse interval. Timing these events can be handled by one of the available 16-bit timers.

3.1 Method - AVR solution

The intervals of the triggers can be calculated on the Ambustim, and by using a timer in its 'Normal mode', the Output Compare Register (OCR) can be set at the length of the trigger pulse so that when the timer matches to OCR the interrupt service routine sets the trigger flag to false for the

period of the inter-pulse interval. This would repeat for the number of pulses at the calculated intervals. To achieve this, the 16 bit Timer 1A was set up,

```
#define CLK_PRESCALE_256 0x04 // Select prescaler settings timer 3:
    f_clk3=7.3728/256 = 28.8 kHz, so t_clk3=35 us
#define RELEASE_TIMERS 0X00
#define RESET_TIMER 0X0000
#define PAUSE_TIMERS 0x81 // Timer Synchronization Mode, prescaler Reset
     Timers
#define enable_Interrupt_1A 0x10 // Only the interrupt bit is set
#define Timer_256_2ms 58 // calculations shown below

// Configure Timer 1A
TCCR1A = CLK_PRESCALE_256;
OCR1A = Timer_256_2ms;
SFIOR = PAUSE_TIMERS;
TCNT1 = RESET_TIMER;
SFIOR = RELEASE_TIMERS;
TIMSK = enable_Interrupt_1A;
// trigger EEG
```

The 16-bit timer means that 2^{16} ticks occur before the overflow flag is set and the timer resets. Since the clock of the microcontroller is set to 7.37MHz, choosing a prescalar of 256 means that every 256 ticks of the clock, correspond to one tick in the counter/timer. Given that the clock of the system is 7.3728MHz this means that one tick corresponds to,

$$Fclk3 = 7.3728/256 = 28.8kHz$$
$$Tclk3 = 34.7us;$$

Using 16 bit timer, a prescalar at 256 and a clock of 7.3728 MHz, time taken for the timer to overflow,

$$OverflowTime = (2^{16} * 256)/7.3728 = 2.3seconds$$

For 2ms pulses, the number of cycles is calculated as,

$$TimerCycles = 2ms/34.7us = 58$$

Using the same method for 5ms and 7ms, corresponds to:

```
// Cycle numbers were tuned during test
#define Timer_256_2ms 58
#define Timer_256_7ms 202
#define Timer_256_5ms 144
```

The next step was to add a new commands to the list,

```
typedef enum {
    // rest of the commands
    TRIGGER_MODE_SQUARE = 'X',
    TRIGGER_MODE_SINE = 'Y',
    TRIGGER_MODE_REGULAR = 'Z'
} Commands;
```

Enumerations were used to communicate the state of the trigger, whether the trigger is on or off, and when a trigger pattern is complete, i.e. all intervals have been iterated through.

```
// Define the enum for Trigger states
typedef enum {
  TRIG_ON,
```

```
  TRIG_OFF,
  TRIG_END,
} TriggerState;
```

Once the pulse intervals were calculated, the ISR was set up (check Appendix B for the full implementation),

```
case TRIG_OFF:
    // Turn on trigger
    OCR1A += Timer_1024_2ms; // add trigger on time
    // other logic

case TRIG_ON:
    // Turn off trigger
    // if sequence does not terminates
    OCR1A += trig_intervals[trigger_count]; // add trigger off time
    // other logic
```

By using this implementation, additionally pulse patterns can be implemented by simply populating the `trig_Interval` array. This helps make the trigger pulse implementation adaptable to other pulse sequences that may later be included in the system.

3.1.1 Regular Trigger

- Command: **X**, Number

    1. Number = trigger number to be sent to EEG amplifier, must be between 0 and 255.

Here, the most basic idea of a trigger is seen, once a trigger event is generated and an S command has been received (prior), the pulse to the patient is initiated and the same trigger is sent to the EEG. This signals to the researcher that the pulse has stimulated the patient and the corresponding EEG is the response to the stimulus.

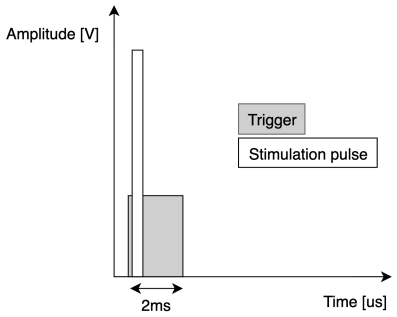

Figure 7: RegularTrigger

3.1.2 Square Wave

- Command: X, Duration, Duty Cycle, Frequency

  1. Sequence Length = sequence duration [ms].
  2. Duty Cycle = duty cycle of square wave modulation in [integer].
  3. Frequency = frequency of square wave modulation [Hz].

Once a trigger command is received, a function to calculate the trigger intervals for a square wave is called. Calculating the number of 2ms pulse, with a 5ms inter-pulse interval, that can fit in the duration of an `on_period` based on the frequency and the duty cycle. The pattern is repeated for the duration of the sequence specified in the parameter passed.

A python script was written on this implementation (check Appendix section C.3) to check the output of the set of parameters.

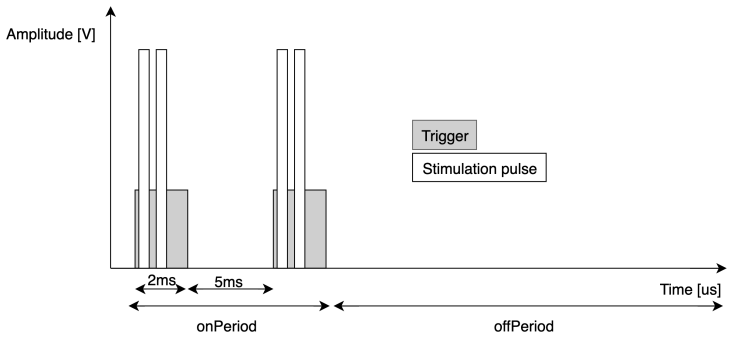

Figure 8: Square Trigger Mode for a double pulse stimulation

3.1.3 Multisine Wave

- Command: Y,Duration, Frequency Offset, Amplitude1, Frequency1, Amplitude2, Frequency2, Amplitude3, Frequency3, Interval Calibration

  1. Duration = sequence duration [ms].
  2. Frequency Offset = frequency offset of multisine [Hz].
  3. Amplitude1,3 = frequency amplitude first sinusoid [Hz].
  4. Frequency1,3 = modulation frequency first sinusoid [Hz].
  5. Interval Calibration (PHI) = phase shift of second (+PHI/12) and third (-PHI/12) frequency component.

A python script was also implemented here to verify the output when taken away from a real time paradigm and employed on an offline/functional paradigm. The script can be found in the Appendix C.4, and serves as a suggestion for a possible C implementation.

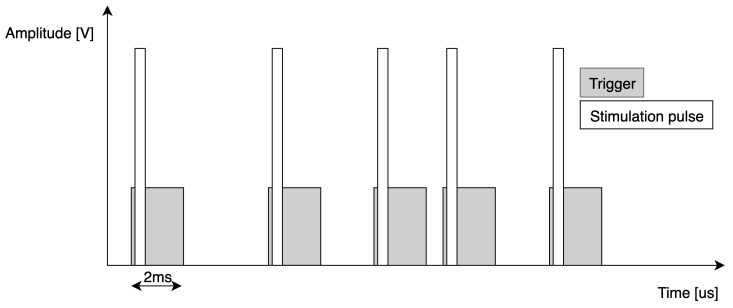

Figure 9: MultiSine Trigger Mode showing variable interpulse intervals for single pulse stimulation

3.2 Code Modularity

By applying the principle of separation of concerns, various aspects of the system were identified and segregated into distinct modules. This entailed relocating the variables that each module depends upon to their corresponding '.h' files. Furthermore, by declaring these variables as static and using getters and setters, they are shielded from unintended external manipulation. Integrating the triggering functionality became a streamlined process, merely involving the addition of its respective '.c' and '.h' files, and adding the new commands to the protocol in the Protocol module.

This was done to clarify the relationship, and interactions, between all the variables and the individual modules.

3.3 Validation checks

The validation of the frequency parameter serves as an essential first check in this context because of the constraints set on the implementation. According to literature, the base frequency in the square wave modulated pulse sequence that allows for a maximum representation of 143Hz. The selection of the 5ms interpulse interval is not arbitrary; it was carefully chosen to limit the effects of peripheral nerve repolarization on measured SSEPs [4].

In the final system design, other validation checks on the different parameters would be necessary, this is to ensure the ranges for a series of parameters are valid and producible by the device's hardware. Some of these parameters are yet to be specified.

4 Validation

To test the implementation, a convenient way of connecting to the Ambustim was developed. A simple python script that discovers the MAC addresses of nearby Bluetooth devices. On the Bluetooth module, the last few characters of the MAC address were printed onto the body. Check the Appendix section C.1 for this script.

Next, a second script was developed for connecting to and communicating with the Ambustim via said Bluetooth module. This script can also be found in the Appendix section C.2.

For performance testing, pin states were toggled corresponding to trigger states, showcasing the improvement over the previous implementations. In the actual system, while the first trigger pulse is emitted to the EEG, its corresponding state in the embedded code is also set to `TRIG_ON`. Subsequent pulses toggle between the states `TRIG_ON` and `TRIG_OFF` in the code.

Assessing runtime memory was challenging due to the lack of simulator support for this chip in Microchip Studio. Print-based debugging was employed for determining cycle counts for tasks like square wave calculations.

4.1 Pulse width

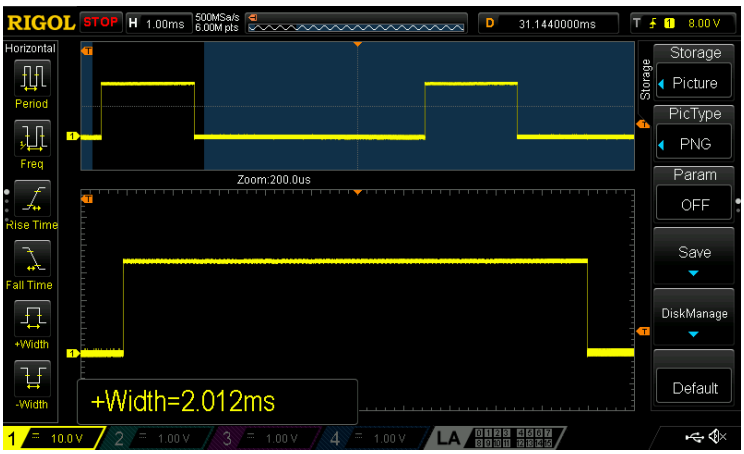

Figure 10: Pulse width accuracy.

4.2 Square wave modulation

The displayed images from the old implementation use the suggested command parameters with the corresponding calibration constants:

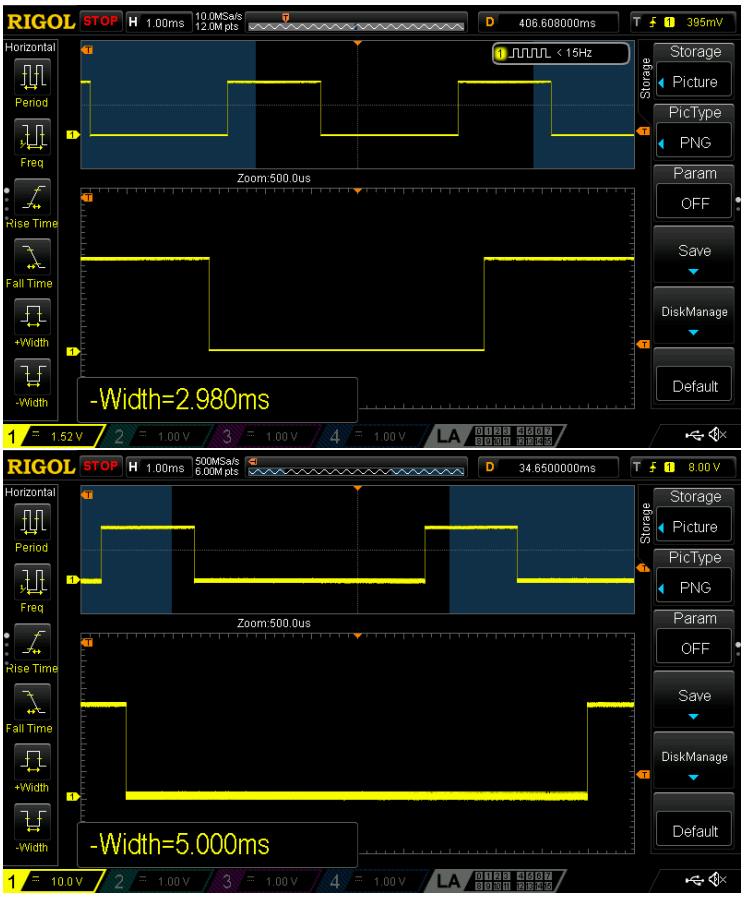

Figure 11: The interpulse interval shows significant improvement over old implementation (top).

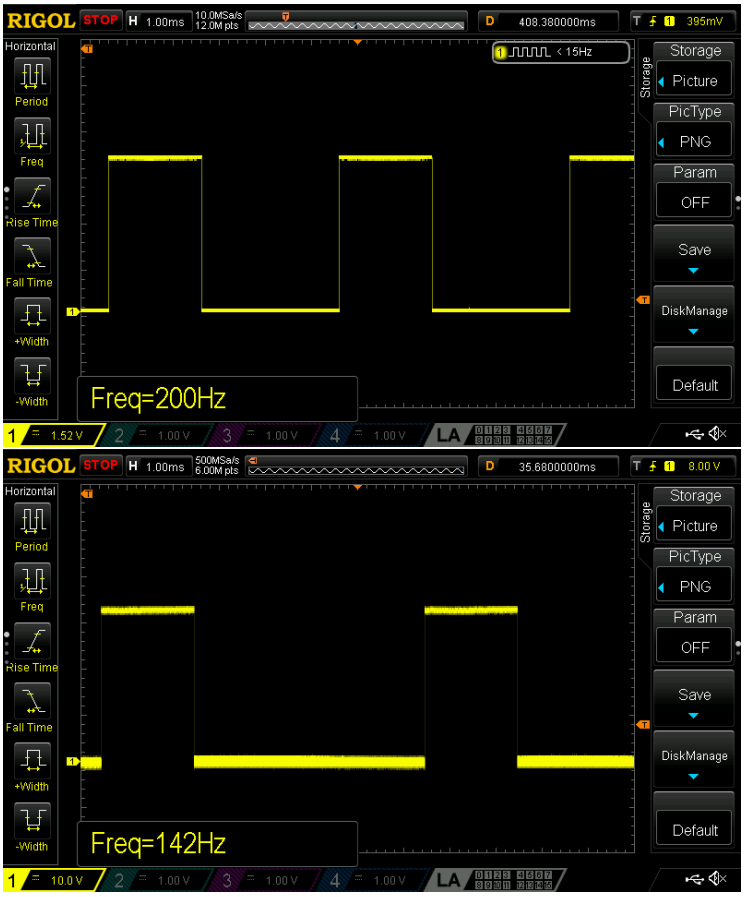

Figure 12: Old implementation (top) shows the frequency of the trigger pulses being at 200Hz instead of the intended ≈142Hz. The new implementation (bottom) displays the reliability and accuracy in using timer interrupts.

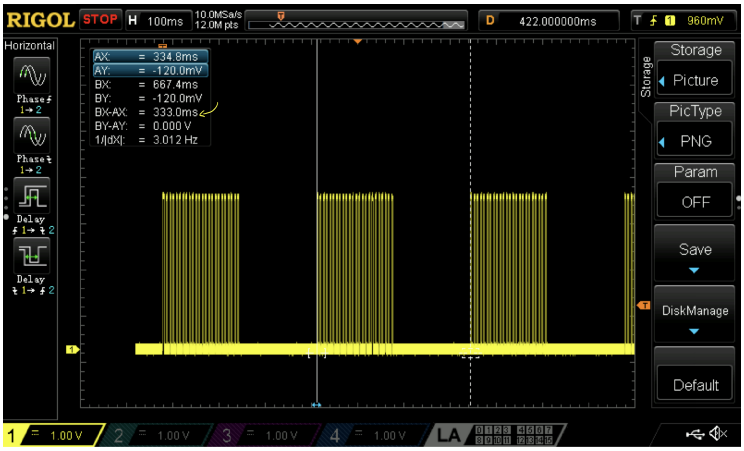

Figure 13: Under the new implementation the duration of the `on_period` + `off_period` matches the intended period of 333ms.

5 Discussion

The software-driven solution demonstrated increased accuracy in the interpulse intervals. The removal for the need of calibration constants streamlines the application of any integer frequency without manual calibration. For the square wave modulated trigger pattern, the floating point calculations were done with 40% of the SRAM still free. Less SRAM might affect performance, as

intermediate calculations reside in the SRAM.

The limited RAM of 1kB put a limit on the size of the arrays that could be used. In the original implementation the arrays sizes were much larger, taking up a lot of the SRAM. For the implementation discussed in the paper, the arrays were sized sufficiently for the device to execute all the commands, and perform adequately.

Arrays were ostensibly sized largely due to the buffer being read within the main loop. To mitigate potential data races caused by UART interrupts, disabling the UART receive flag (RXEN1 or RXCIE1) is recommended when accessing the buffer. For double-pulse (stimulation) tests, a buffer size of 20 proved adequate, rendering the original 200 (buffer size) unnecessary.

Memory was conserved by repeating patterns through a counter variable in the square wave as opposed to storing the additional intervals. While this optimized memory usage, it added complexity to the square wave modulation, resulting in an extended ISR. For a Multisine modulation scenario, increasing available memory would simplify the implementation and shorten the ISR.

Trigger pulse patterns require floating point calculations. Using a microcontroller equipped with a Floating Point Unit (FPU) would be advised, as the current calculation for the 3Hz square wave modulation case demands 70*256 clock cycles to complete the floating point calculations. This duration would extend for multisine operations due to the involvement of more floating point values.

Regardless of the additional SRAM, an 8 bit system, in the absence of an FPU, might have similar performance characteristics to the ATMega162. This is because mere memory increments might not significantly curtail execution cycles. Considering a 16 bit system would improve floatin point caluclations, but an FPU would offer the most efficiency. Moreover, such a transition might also offer power conservation benefits, a critical consideration for battery-dependent devices like the Ambustim.

6 Conclusion

This paper introduces an optimized design for generating modulated triggering patterns in the *Ambustim*, an electrical stimulation device employed for nociceptive system studies and pain perception analysis. The focus was to implement pattern generating functionality into the Ambustim's firmware, thereby reducing system complexity by eliminating the need for calibration constants, and making the system more accessible for practical usage beyond lab settings. By employing hardware interrupts for triggering patterns directly within the firmware, timing accuracy was achieved.

The design was tested for square wave and multisine wave modulated triggering patterns through Python simulations, validating the square wave modulation through pin read-outs, and providing a recommendation on an implementation for multisine modulation. The lack of a Floating Point Unit (FPU) remains a bottleneck for more complex tasks like Multisine triggering, suggesting the potential benefits of upgrading to a more advanced microcontroller for future work.

The outcomes showed a considerable improvement in the accuracy over the previous design, which relied on software loops and calibration constants. The current approach not only eliminates the need for manual calibration but also makes it feasible to apply uncalibrated integer frequency for stimulation.

In conclusion, this research contributes a meaningful step toward simplifying the `Ambustim`'s setup and enhancing its accuracy and versatility. Thus aiding in more effective research into nociceptive systems and pain perception.

A Appendix I

A.1 Trigger

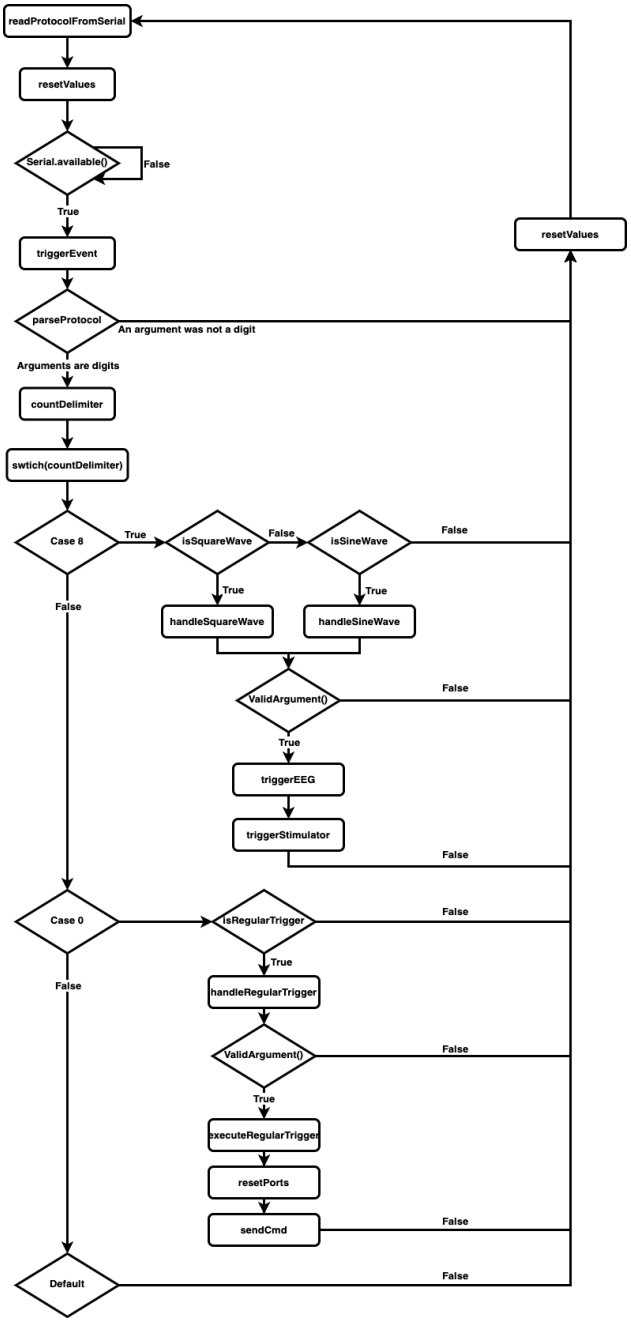

Figure 14: Trigger Box code

B Appendix II

```
#include "TriggerModes.h"
#include "Protocol.h"

// Trigger parameters
static volatile uint16_t trig_intervals[MAX_TRIGGER_PATTERN],
    trigger_counter, trig_intervals_array_size;
static volatile uint16_t repetitions, repetitions_counter;
```

```
7  static volatile TriggerState trig, old_trig;
8
9  // Square Wave modulated pulse sequence, command:    X, <DURATION>, <
       DUTYCYCLE>, <FREQ>
10 static volatile uint16_t total_sequence;
11 static volatile double duty_cycle;
12 static volatile uint32_t on_period;
13 static volatile uint32_t off_period;
14
15
16 // Alternate between trigger states until the end of the specified
       sequence length (total_sequence)
17
18 SIGNAL(TIMER1_COMPA_vect) {
19     switch(trig){
20         case TRIG_OFF:
21             // Turn on trigger flag
22             sbi(PORTA, 5);
23             trig = TRIG_ON;
24             OCR1A += Timer_256_2ms; // add trigger on time
25             break;
26
27         case TRIG_ON:
28             // Turn off trigger flag
29             cbi(PORTA, 5);
30
31             // Check if sequence terminates
32             if(trigger_counter < trig_intervals_array_size){
33                 OCR1A += trig_intervals[trigger_counter]; // add trigger
       off time
34                 trigger_counter++;
35             }
36
37             // Check for completion after incrementing TriggerCount
38             if(trigger_counter >= trig_intervals_array_size){
39                 trigger_counter = 0;
40
41                 repetitions_counter++;
42
43                 if(repetitions_counter >= repetitions){
44                     // Sequence is done
45                     trig = TRIG_END;
46                     break;
47                 }
48
49                 trig = TRIG_OFF;
50             } else{
51                 // Continue sequence
52                 trig = TRIG_OFF;
53             }
54             break;
55
56         case TRIG_END:
57             // Handle end condition immediately
58             TIMSK = ZERO; // Disable interrupts from timer1 compA and
       compB
59             TCCR1A = ZERO; // Stop clocksource timer 1
60             break;
61
62         default:
63             // Reserved for unexpected Trig states.
64             break;
65     }
```

```
}

// Initializes trigger-related variables.
// This function resets the variables responsible for the intervals, including the states of the triggers. Used to ensure that previous values do not interfere with the new sequence.
void initialize_trigger() {
    int i;
    // Initialize variables for trigger intervals
    for(i = 0; i < MAX_TRIGGER_PATTERN; ++i){
        trig_intervals[i] = 0;
    }

    trigger_counter = 0;
    trig_intervals_array_size = 0;
    trig = TRIG_OFF;
    old_trig = TRIG_OFF;
}

//Configures Timer 1 for handling the trigger sequences.
void setup_trig_timer() {
    // Configure Timer 1A
    TCCR1A = CLK_PRESCALE_256;
    OCR1A = Timer_256_2ms;
    SFIOR = PAUSE_TIMERS;
    TCNT1 = RESET_TIMER;
    SFIOR = RELEASE_TIMERS;
    TIMSK = enable_Interrupt_1A;

    sbi(PORTA, 5);
    trig = TRIG_ON;
    old_trig = TRIG_OFF;
}

// Calculates the intervals for a square wave pulse sequence based on input parameters. Given the total sequence time, frequency and duty cycle, this function calculates the on and off periods for a square wave. It then translates these durations into intervals that the system should wait between triggers.The function also computes how many times the trigger sequence should be repeated to match the desired total sequence duration
void calculate_square_wave_intervals() {
    // Play this many triggers
    trig_intervals_array_size = on_period / (Timer_256_7ms * TIMER_256_to_us);
    off_period = (on_period + off_period) / TIMER_256_to_us - (trig_intervals_array_size * Timer_256_7ms);

    // 5ms between each trigger
    for(int i = 0; i < trig_intervals_array_size; i++) {
        trig_intervals[i] = Timer_256_5ms;
    }

    // if on_period fits more than once in the total sequence
    uint16_t temp1 = total_sequence;
    uint16_t temp2 = (trig_intervals_array_size * Timer_256_7ms * ms_to_us * TIMER_256_to_us ) + (off_period * ms_to_us * TIMER_256_to_us);
    repetitions = ((double) temp1 / temp2) - 1;
    repetitions_counter = -1;
    if(repetitions >= 1) {
        trig_intervals[trig_intervals_array_size] = off_period;
        trig_intervals_array_size += 1;
```

```
    }
}
```

C Appendix III

C.1 Python script for discovering the Ambustim

```
import subprocess
import re

def discover_devices():
    result = subprocess.run(['system_profiler', 'SPBluetoothDataType'],
    stdout=subprocess.PIPE, text=True)

    devices = re.findall(r'Address: (.*)', result.stdout)

    print("Found {} devices.".format(len(devices)))

    for addr in devices:
        print("  {}".format(addr))

discover_devices()
```

C.2 Python script for communication with the Ambustim

```
import socket
import time
import threading

MACAddress = '00:01:95:0C:C2:61'  # NociTrack
port = 1
s = socket.socket(socket.AF_BLUETOOTH, socket.SOCK_STREAM, socket.
    BTPROTO_RFCOMM)
s.connect((MACAddress, port))
print("Connection established")
s.settimeout(4)
time.sleep(1)
btReadSize = 4000

def receiver(sock, btReadSize):
    while True:
        data = sock.recv(btReadSize)  # read bluetooth data
        if data:
            text = data.decode('UTF-8')
            print("", text)
        else:
            print("No data read, re-connecting...")
            sock.connect((MACAddress, port))

# Start the receiver thread
receiver_thread = threading.Thread(target=receiver, args=(s, btReadSize)
    )
receiver_thread.start()

# Start Up F command <,,> , sets NUMCHANNELS, MAXPATTERN, AD2mA and
    TU2ms
s.send(bytes('F,0,0,0,0\0', 'UTF-8'))
print("sent data: F,0,0,0,0")
time.sleep(1)  # Add delay here
# Sets Version1 Version2 and SerNo
s.send(bytes('V,0,0,0\0', 'UTF-8'))
```

```
time.sleep(1)  # Add delay here
s.send(bytes('M,1,1\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('C,1,1,0\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('I,2000,2000\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('P,1,1\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('A,60,60\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('a,0,0\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('W,480,480\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('w,400,400\0', 'UTF-8'))
time.sleep(1)  # Add delay here
s.send(bytes('S, 0 , 1, 1000\0', 'UTF-8'))
time.sleep(1)  # Add delay here
# 1 sec sequence with 50% duty cycle and 3Hz frequency
s.send(bytes('X,1000,50,3\0', 'UTF-8'))
```

C.3 Python script for simulating Square Wave Modulated pulse sequence

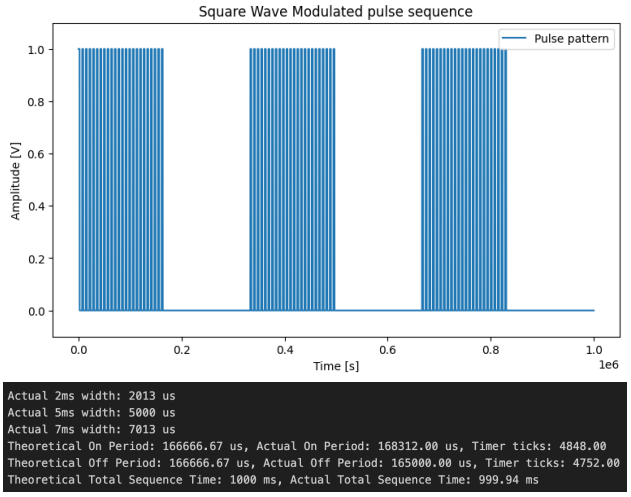

Figure 15: Square wave modulated pulse sequence python code used for testing outcomes of implementation

```
import matplotlib.pyplot as plt

# Parameters
total_sequence_ms = 1000
duty_cycle_percent = 50
freq_hz = 3

# Defines
PULSE_AMP = 1
Freq_CPU = 7372800
PRE_SCALERRR = 256
T_256_2ms = 58
T_256_5ms = 144
T_256_7ms = 202
TIMER_TO_US = PRE_SCALERRR * 1000000 // Freq_CPU
```

```
16
17 # Time
18 Timer_256_2ms = (T_256_2ms * PRE_SCALERRR * 1000000) // Freq_CPU  # 2ms
       in us
19 Timer_256_5ms = (T_256_5ms * PRE_SCALERRR * 1000000) // Freq_CPU  # 5ms
       in us
20 Timer_256_7ms = (T_256_7ms * PRE_SCALERRR * 1000000) // Freq_CPU  # 7ms
       in us
21
22 # Print widths
23 print(f"Actual 2ms width: {Timer_256_2ms} us")
24 print(f"Actual 5ms width: {Timer_256_5ms} us")
25 print(f"Actual 7ms width: {Timer_256_7ms} us")
26
27 # Case: Trig_Mode_Square
28 duty_cycle = duty_cycle_percent / 100.0
29 on_period_theoretical = (1.0 / freq_hz) * duty_cycle * 1000000  # in us
30 off_period_theoretical = (1.0 / freq_hz) * (1.0 - duty_cycle) * 1000000
       # in us
31
32 # Calculate trigger intervals for square wave mod
33 num_pulses_on_period = int(on_period_theoretical / Timer_256_7ms) + 1
34 on_period_actual = Timer_256_7ms * (num_pulses_on_period)
35 off_period_actual = (on_period_theoretical + off_period_theoretical) - (
       num_pulses_on_period * Timer_256_7ms )
36 off_period_actual = int(off_period_actual/(PRE_SCALERRR*1000000/Freq_CPU
       )) * (PRE_SCALERRR*1000000/Freq_CPU)
37 total_sequence_actual_ms = (on_period_actual + off_period_actual) *
       freq_hz / 1000
38
39 # Validate outcomes
40 print(f"Theoretical On Period: {on_period_theoretical:.2f} us, Actual On
       Period: {on_period_actual:.2f} us, Timer ticks: {(T_256_7ms*
       num_pulses_on_period):.2f}")
41 print(f"Theoretical Off Period: {off_period_theoretical:.2f} us, Actual
       Off Period: {off_period_actual:.2f} us, Timer ticks: {(
       off_period_actual/(PRE_SCALERRR*1000000/Freq_CPU)):.2f}")
42 print(f"Theoretical Total Sequence Time: {total_sequence_ms} ms, Actual
       Total Sequence Time: {total_sequence_actual_ms:.2f} ms")
43
44 # Calculate total sequence
45 sequence = []
46 for _ in range(num_pulses_on_period):
47     sequence += [PULSE_AMP] * int(Timer_256_2ms) + [0] * int(
       Timer_256_5ms)
48 sequence += [0] * int(off_period_actual)
49
50 # Repeat
51 sequence = sequence * freq_hz
52
53 # Plotting
54 plt.plot(sequence)
55 plt.ylim(-0.1, PULSE_AMP + 0.1)
56 plt.title('Square Wave Modulated pulse sequence')
57 plt.xlabel('Time [us]')
58 plt.ylabel('Amplitude')
59 plt.show()
```

C.4 Python script for simulating Multisine Wave Modulated pulse sequence

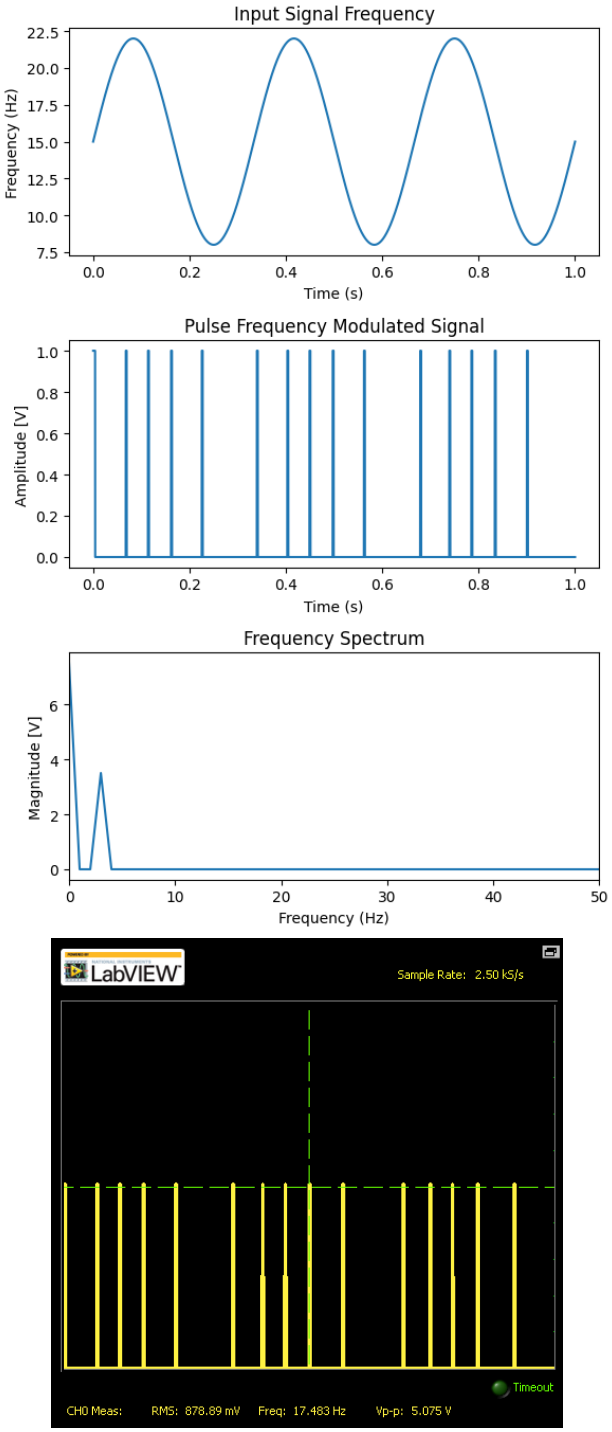

Figure 16: Multisine wave modulated pulse sequence python code used for testing outcomes of implementation. The arduino implementation against the one implemented in python are aligned. Emitting the first pulse immediately, and then using the computed instantaneous frequencies for the subsequent pulses. The frequency spectrum shows the 3Hz magnitude of half the amplitude with the appropriate phase shift. This is because only the first half of the spectrum is being plotted

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft
```

```
4
5 # Hardware based
6 Freq_CPU = 7372800
7 PRE_SCALER = 256
8 T_256_2ms = 58
9
10 # Constants
11 PI = np.pi
12 RES = 100000000
13 PULSE_AMPLITUDE = 1
14 MAX_FREQ_FFT = 50
15
16 # Parameters
17 SEQ_DUR = 1  # Total time in seconds
18 OFFSET = 15.0 # Frequency offset
19 AMP1 = 7.0 # Amplitude for each sin component
20 AMP2 = 0.0
21 AMP3 = 0.0
22 FREQ1 = 3.0 # Frequency for each sin component
23 FREQ2 = 3.0
24 FREQ3 = 10.0
25 PHI = 8 # Phase shift
26
27 # Time
28 Timer_256_2ms = (T_256_2ms * PRE_SCALER * 1000000) // Freq_CPU  # 2ms in
       us
29 RESOLUTION = Timer_256_2ms / RES  # Time resolution in seconds, based on
       timer
30 pulse_width = Timer_256_2ms / 1000000  # Pulse width in seconds, 2ms
31 T = np.arange(0, SEQ_DUR, RESOLUTION)
32 signal_pulses = np.zeros_like(T)
33
34 # Instantaneous period calculation
35 freq = OFFSET + AMP1 * np.sin(2.0 * PI * FREQ1 * T) + AMP2 * np.sin(2.0
       * PI * FREQ2 * T + (PHI * PI / 12.0)) + AMP3 * np.sin(2.0 * PI *
      FREQ3 * T - (PHI * PI / 12.0))
36 ipi = 1 / freq
37
38 # Set the first pulse at t=0
39 start_pulse = 0
40 end_pulse = int(pulse_width / RESOLUTION)
41 signal_pulses[start_pulse:end_pulse] = PULSE_AMPLITUDE
42
43 # Start with the time of the first pulse
44 next_pulse_time = pulse_width
45 current_time = 0
46
47 # Generate pulses based on the signal amplitude
48 for i in range(len(T)):
49 current_time += RESOLUTION
50 if current_time >= next_pulse_time:
51 start = i
52 end = i + int(pulse_width / RESOLUTION)
53 if end < len(T):
54     signal_pulses[start:end] = PULSE_AMPLITUDE  # Add 2ms pulse width
55 next_pulse_time += ipi[i]  # Update the next pulse time
56
57 # Compute the FFT of the signal
58 fft_output = np.fft.fft(freq)
59 fft_magnitude = np.abs(fft_output)[:len(T)//2] / len(T)
60 fft_magnitude[0] /= 2  # Do not double the DC component
61 fft_frequency = np.fft.fftfreq(len(T), d=RESOLUTION)[:len(T)//2]
62
```

```
63 # Plot
64 fig, axs = plt.subplots(3, 1, figsize=(6, 9))  # Create 4 subplots
65
66 # Waveform
67 axs[0].plot(T, freq)
68 axs[0].title.set_text('Input Signal Frequency')
69 axs[0].set_xlabel('Time (s)')
70 axs[0].set_ylabel('Frequency (Hz)')
71
72 # PFM
73 axs[1].plot(T, signal_pulses)
74 axs[1].title.set_text('Pulse Frequency Modulated Signal')
75 axs[1].set_xlabel('Time (s)')
76 axs[1].set_ylabel('Amplitude [V]')
77
78 # Frequency Spectrum
79 axs[2].plot(fft_frequency, fft_magnitude)
80 axs[2].title.set_text('Frequency Spectrum')
81 axs[2].set_xlabel('Frequency (Hz)')
82 axs[2].set_ylabel('Magnitude [V]')
83 axs[2].set_xlim([0, MAX_FREQ_FFT])
84
85 plt.tight_layout()
86 plt.show()
```

References

[1] B. van den Berg, L. Vanwinsen, G. Pezzali, and J. R. Buitenweg, "Observation of nociceptive detection thresholds and cortical evoked potentials: Go/no-go versus two-interval forced choice," *Atten Percept Psychophys*, vol. 84, pp. 1359–1369, May 2022.

[2] B. van den Berg, R. J. Doll, A. L. H. Mentink, P. S. Siebenga, G. J. Groeneveld, and J. R. Buitenweg, "Simultaneous tracking of psychophysical detection thresholds and evoked potentials to study nociceptive processing," *Behav Res Methods*, vol. 52, Aug 2020.

[3] B. van den Berg and J. R. Buitenweg, "Observation of nociceptive processing: Effect of intraepidermal electric stimulus properties on detection probability and evoked potentials," *Brain Topogr*, vol. 34, pp. 139–153, Mar 2021.

[4] B. v. d. Berg, *Combined Psychophysical and Neurophysiological Tools for Mechanism-Based Observation of Impaired Nociceptive Processing*. PhD thesis, University of Twente, 2022.

[5] B. van den Berg, R. J. Doll, A. L. H. Mentink, P. S. Siebenga, G. J. Groeneveld, and J. R. Buitenweg, "Multisine frequency modulation of intraepidermal electric pulse sequences: A novel tool to study nociceptive processing," *Journal of Neuroscience Methods*, vol. 353, p. 109106, 2021.

[6] A. Corporation, "8-bit avr microcontroller with 16k bytes in-system programmable flash." `https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-2513-8-bit-AVR-Microntroller-ATmega162_Datasheet.pdf`.