Developing a Configurable Metronome with the MSP430 Microcontroller

PHYS 319 Final Project

By Robbie Laughlen

Abstract

The project titled "Developing a Configurable Metronome with the MSP430 Microcontroller" aimed to design and implement a digital metronome using the MSP430 microcontroller. The metronome allowed users to customise the tempo and time signature settings based on their musical needs. The results of the experiment showed that the metronome was able to accurately generate the desired tempo and beat count, as confirmed through comparisons with a reference metronome. The digital nature of the metronome provided precise and consistent timing, making it a reliable tool for musicians. The project concluded that the customizable metronome offers advantages over traditional mechanical metronomes in terms of flexibility, accuracy, and portability. Despite some limitations, the metronome shows potential for further improvements and applications in the field of music technology.

Introduction

The concept of a metronome, a device that helps musicians maintain a consistent tempo and rhythm, has been widely used in music practice and performance for centuries. Traditional mechanical metronomes offer limited options for adjusting the tempo and time signature, and their accuracy may vary. With the advancement of digital technology, microcontrollers have been increasingly utilised to create customizable and precise metronomes. In this project, the aim was to design and implement a digital metronome using the MSP430 microcontroller, which would allow users to customise the tempo and time signature settings based on their musical needs. This project builds upon previous work in the field of music technology and seeks to overcome limitations of traditional metronomes by leveraging the capabilities of microcontrollers to provide a versatile and accurate tool for musicians. The motivation behind this project is to create a metronome that offers greater flexibility, accuracy, and portability, and can potentially find applications in music practice, performance, and education.

Theory

The design and implementation of the customizable metronome using the MSP430 microcontroller involves several key concepts. The MSP430 microcontroller is a low-power microcontroller that provides a range of functionalities for digital signal processing, including generating clock signals, reading user input, and driving output devices.

One important aspect of the metronome is repetition of a pulse width modulated signal that can be sent out at regular intervals. The MSP430 microcontroller relies on this Pulse Width Modulation (PWM) to generate a stable, accurate and adjustable signal. The frequency of the note sent to the metronome depends on the PWM, with higher frequencies corresponding to the primary beat, and lower frequencies corresponding to the secondary beats.
Another essential aspect of the metronome is the calculation of the beat count based on the time signature input by the user. The time signature typically consists of two numbers, such as 4/4 or 3/8, where the numerator represents the number of beats per measure and the denominator indicates the type of note that receives one beat. The MSP430 microcontroller calculates the beat count based on the time signature input and adjusts the metronome output accordingly.

The user input is crucial for setting the tempo and time signature of the metronome. The MSP430 microcontroller interfaces with buttons or switches connected to its input pins to read the user's input. The microcontroller uses digital signal processing

techniques to debounce and interpret the user input, allowing for customization of the metronome settings.

The output of the metronome is generated through an output device, such as a piezo buzzer or a speaker, which produces audible beeps to represent the beats of the metronome. The MSP430 microcontroller drives the output device through its output pins, generating the appropriate frequency and duration of the beeps based on the calculated tempo and beat count.

Understanding these theoretical concepts is essential for the successful design and implementation of the customizable metronome using the MSP430 microcontroller, and forms the basis for the subsequent apparatus, results, and discussion sections of this project.

Apparatus

Components

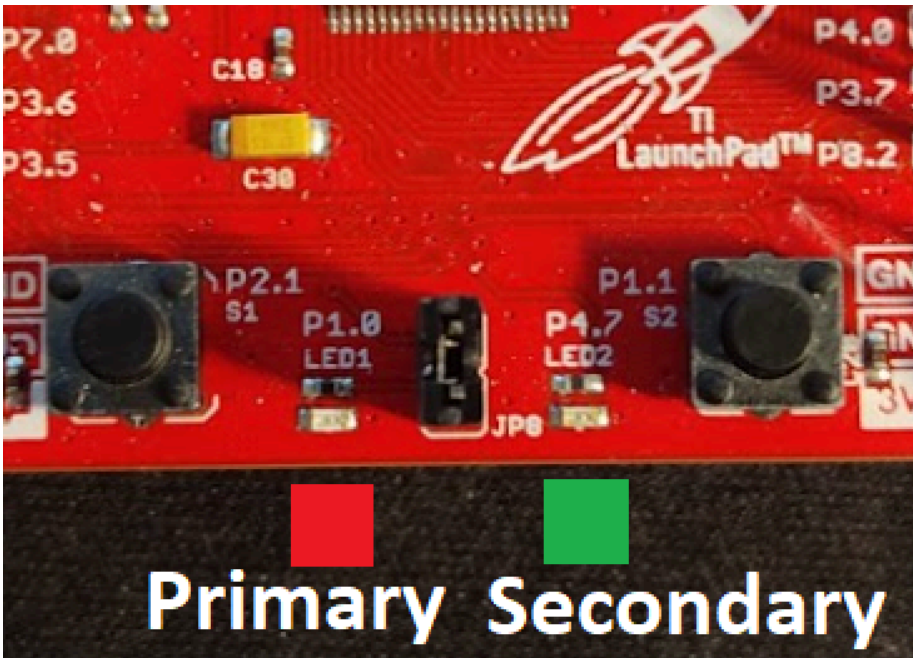
The apparatus used in this project to create a customizable metronome using the MSP430 microcontroller consists of several components, including the microcontroller, input devices, output devices, and power source.

- **MSP430 Microcontroller**: The MSP430 microcontroller is the main control unit of the metronome, responsible for generating the signal, reading user input, calculating the beat count, and driving the output device. The microcontroller used in this project is the MSP430F5529, which is a low-power microcontroller with integrated peripherals such as GPIO (General-Purpose Input/Output) pins, timers, and UART (Universal Asynchronous Receiver/Transmitter) for communication.

- **Input Devices**: The input devices are used to customise the tempo and time signature settings of the metronome. In this project, buttons or switches are used as input devices, connected to the GPIO pins of the MSP430 microcontroller. These buttons or switches allow the user to adjust the tempo of the metronome.

- **Output Devices**: The output device is responsible for generating the audible beats of the metronome. In this project, a piezo buzzer is used as the output

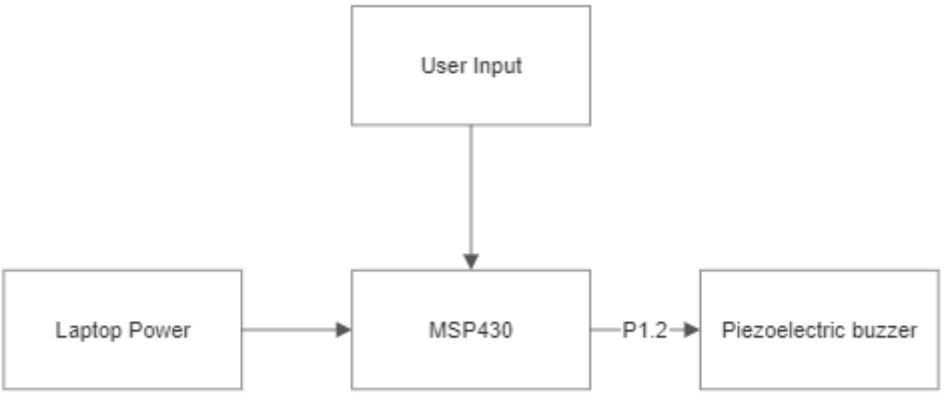
device, connected to one of the GPIO pins of the MSP430 microcontroller. The MSP430 microcontroller generates the appropriate frequency and duration of the beeps based on the calculated tempo and beat count, driving the piezo buzzer to produce the beats. The LEDs on P1.0 and P4.7 flash based on the primary and secondary beats of the metronome.



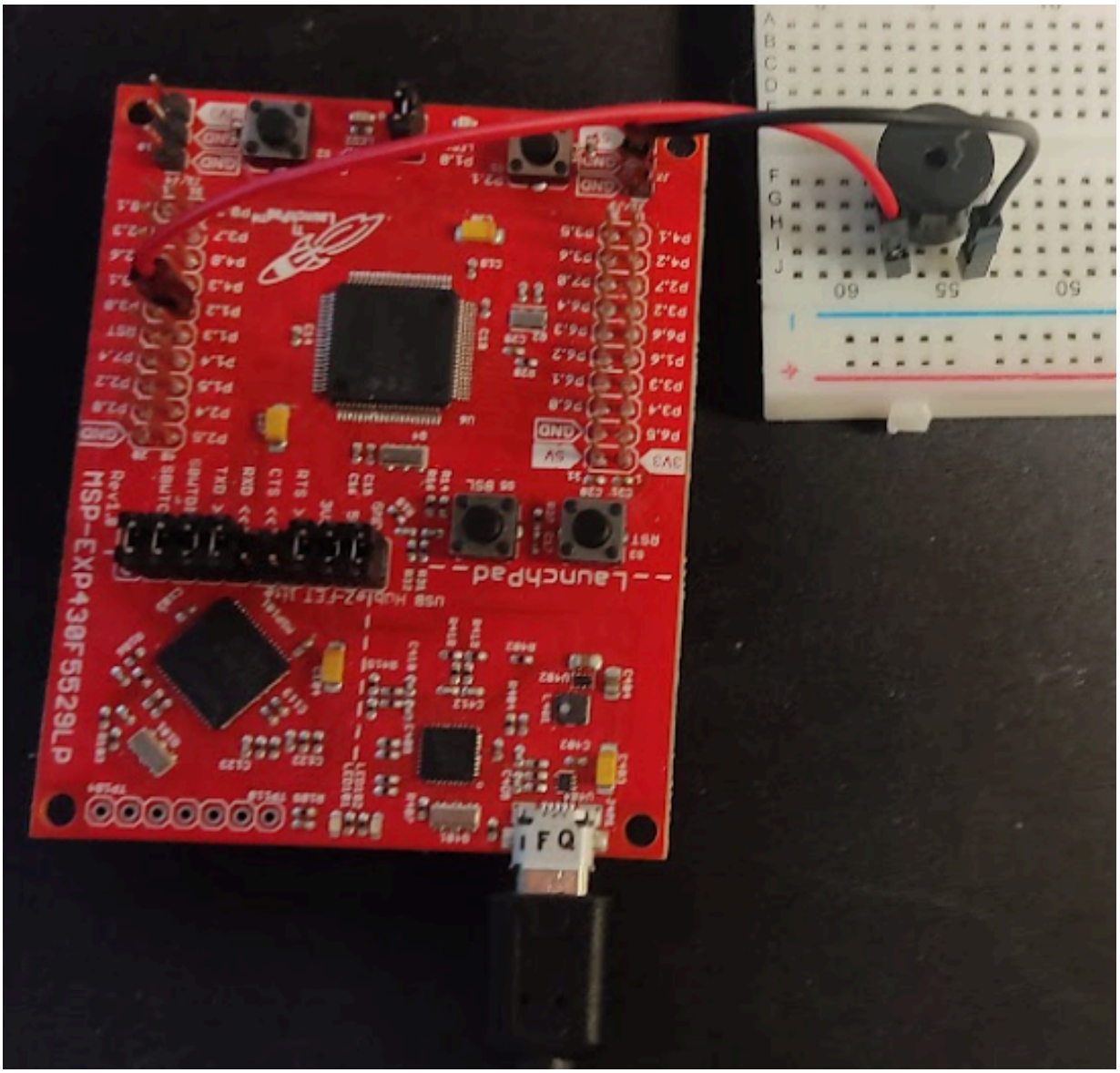
- **Power Source**: The apparatus requires a power source to operate. In this project, a laptop is connected to the MSP430 microcontroller to ensure constant power. The breadboard was used as a basic framework for the circuit and did not require power.

Functional Block Diagram

A functional block diagram is used to illustrate the overall operation of the metronome apparatus. The diagram shows the interconnection of different functional blocks, including the microcontroller, input devices, output device, and power source, and their interactions in generating the metronome beats based on user input.



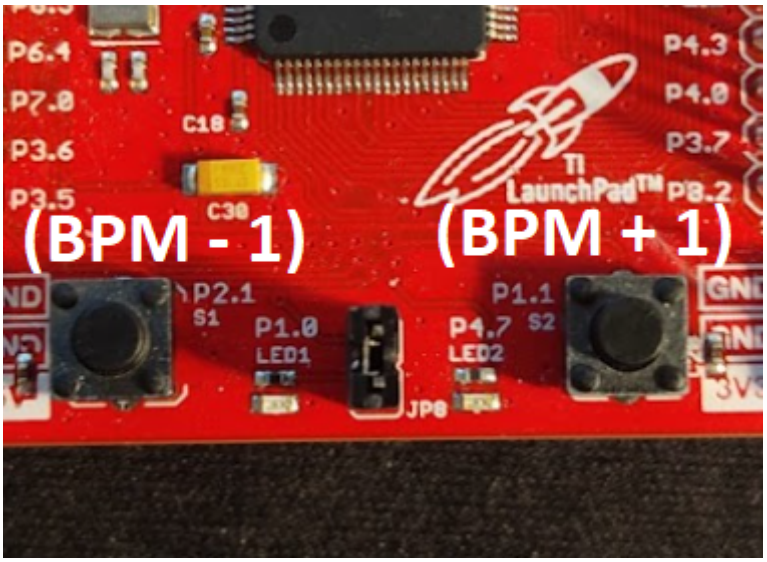
Full Circuit Setup



Description of Operation

The customizable metronome operates as follows:

1. The user adjusts the time signature and base starting BPM within the code.
2. The microcontroller generates a stable and accurate signal based on the delay values input, which determines the tempo of the metronome.
3. Based on the calculated tempo and beat count, the microcontroller generates the appropriate frequency and duration of the beeps, driving the piezo buzzer to produce the beats.
4. The metronome produces audible beats through the piezo buzzer, allowing the user to practise or perform music with precise timing.
5. The user can then adjust the tempo of the metronome using the P1.1 button to increase BPM by 1, and the P2.1 button to decrease the tempo by 1.
6. The LEDs then sync with the primary and secondary beats of the metronome, with more secondary beats when there is an increase in time signature



The apparatus is designed to be compact and portable, allowing for easy use in various musical settings. The block diagram and electrical schematics provide a clear understanding of the functional components and their interactions in the metronome apparatus, facilitating the replication and further development of the project.

Code Explanation

Macro Definitions

```
#define BEEP_PIN BIT2   // Pin for beep output
#define LED1_PIN BIT0   // Pin for LED1 output
#define LED2_PIN BIT7   // Pin for LED2 output
#define BEATS 4         // Number of beats
#define MIN_BPM 30      // Minimum BPM
#define MAX_BPM 240     // Maximum BPM
#define TICK_RATE 60000 / BPM   // Metronome tick rate in
milliseconds
#define PWM_FREQUENCY1 2093.005 // Frequency of Tone 1 in Hz (c)
#define PWM_FREQUENCY2 1567.98 // Frequency of Tone 2 in Hz (g)
#define DUTY_CYCLE1 500     // Duty cycle of Tone 1 in tenths of
percent (0 to 1000)
#define DUTY_CYCLE2 250    // Duty cycle of Tone 2 in tenths of
percent (0 to 1000)
```

These macro definitions define various parameters used in the code. They define the pins used for beep output (BEEP_PIN), LED1 output (LED1_PIN), and LED2 output (LED2_PIN) as well as the number of beats per measure (BEATS), the minimum and maximum beats per minute (MIN_BPM and MAX_BPM), the metronome tick rate in milliseconds (TICK_RATE), the frequencies of two tones used in the metronome (PWM_FREQUENCY1 and PWM_FREQUENCY2), and the duty cycles of the two tones (DUTY_CYCLE1 and DUTY_CYCLE2).

Function Definition

```
void delay_ms(unsigned int ms)
{
    unsigned int i;
    for (i = 0; i < ms; i++)
    {
```

```
        __delay_cycles(1000);   // Delay 1000 cycles at 1MHz clock
    }
}
```

This function is used to introduce a delay in milliseconds. It uses a loop that iterates for the specified number of milliseconds and delays the execution of the code by 1000 cycles at the clock frequency of 1 MHz using the __delay_cycles() function.

Main Function

```
int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    P1DIR |= BEEP_PIN;   // Set LED pin as output
    P1DIR |= LED1_PIN;   // Set LED pin as output
    P1OUT &= ~LED1_PIN;   // Initialize LED pin to low
    P4DIR |= LED2_PIN;   // Set LED pin as output
    P4OUT &= ~LED2_PIN;   // Initialize LED pin to low

    P1DIR &= ~BIT1;     // Set P1.1 as input
    P1REN |= BIT1;      // Enable pull-up resistor on P1.1
    P1OUT |= BIT1;      // Set pull-up resistor on P1.1
    P1IES |= BIT1;      // Set P1.1 to trigger on falling edge
    P1IE |= BIT1;       // Enable interrupt for P1.1

    TA0CCR0 = 1000 - 1;                  // PWM Period
    TA0CCTL1 = OUTMOD_7;                 // CCR1 reset/set
    TA0CCR1 = (DUTY_CYCLE1 * TA0CCR0) / 1000; // CCR1 PWM duty cycle
    TA0CCTL2 = OUTMOD_7;                 // CCR2 reset/set
    TA0CCR2 = (DUTY_CYCLE2 * TA0CCR0) / 1000; // CCR2 PWM duty cycle
    TA0CTL = TASSEL_2 + MC_1 + TACLR;    // SMCLK, Up mode, Clear TAR

    P2DIR &= ~BIT1;     // Set P2.1 as input
    P2REN |= BIT1;      // Enable pull-up resistor on P2.1
    P2OUT |= BIT1;      // Set pull-up resistor on P2.1
    P2IES |= BIT1;      // Set P2.1 to trigger on falling edge
    P2IE |= BIT1;       // Enable interrupt for P2.1

    __bis_SR_register(GIE); // Enable global interrupts

    while (1)
```

```
    {
        P1OUT ^= LED1_PIN;   // Toggle LED pin
        P1SEL |= BIT2; // Pin 1.2 selected as PWM
        TA0CCR0 = 1000000 / PWM_FREQUENCY1 - 1; // Update PWM period for
Tone 1
        TA0CCR1 = (DUTY_CYCLE1 * TA0CCR0) / 1000; // Update PWM duty
cycle for Tone 1
        delay_ms(50);   // Delay for tick rate
        P1SEL &= ~BIT2; // Pin 1.2 selected as PWM
        P1OUT ^= LED1_PIN;   // Toggle LED pin

    //   if ((BPM >= MAX_BPM) || (BPM <= MIN_BPM))
    //   {
    //       P6OUT &= LIMLED;   // Make LED high
    //   }

        delay_ms(TICK_RATE);   // Delay for tick rate

        int i;
        for (i = 1; i < BEATS; ++i)
        {
            P4OUT ^= LED2_PIN;   // Toggle LED pin
            //P1OUT ^= BEEP_PIN;   // Toggle BEEP pin
            P1SEL |= BIT2; // Pin 1.2 selected as PWM
            TA0CCR0 = 1000000 / PWM_FREQUENCY2 - 1; // Update PWM period
for Tone 2
            TA0CCR2 = (DUTY_CYCLE2 * TA0CCR0) / 1000; // Update PWM duty
cycle for Tone 2
            delay_ms(50);   // Delay for tick rate
            P1SEL &= ~BIT2; // Pin 1.2 selected as PWM
            P4OUT ^= LED2_PIN;   // Toggle LED pin

            delay_ms(TICK_RATE);   // Delay for tick rate
        }
    }

    return 0;
}
```

Watchdog Timer Configuration: The first line of the main function disables the Watchdog Timer (WDT) by setting the WDTCTL register to WDTPW + WDTHOLD.

The Watchdog Timer is a hardware timer that resets the microcontroller if it is not periodically reset. In this code, we disable it to prevent any unintended resets.

Clock Configuration: The next few lines configure the microcontroller's clock. The DCO (Digitally Controlled Oscillator) is configured to operate at 1 MHz, assuming a DCO frequency of 1 MHz. The DCOCTL register is set to select the lowest DCOx and MODx settings, and the BCSCTL1 register is set to configure the DCO range to 1 MHz.

Button Input Configuration: The next line of code configures Pin 1.1 of Port 1 (P1.1) as an input for the button. The P1DIR register is configured to clear the BIT1 position, setting P1.1 as an input.

Button Interrupt Configuration: The next lines configure the button to generate an interrupt on the falling edge. The P1IES register is set to configure the interrupt edge selected for P1.1 as falling edge (i.e., when the button is pressed). The P1IFG register is cleared to reset any pending interrupt flags for Port 1. The P1IE register is set to enable interrupts for Port 1 (P1), specifically for P1.1.

Timer Configuration: The next lines configure Timer A0 (TA0) as the metronome timer. The TACCTL0 register is set to configure TA0CCR0 as the compare mode, which generates an interrupt when the value in TA0CCR0 is reached. The TA0CCR0 register is set to a value corresponding to the initial BPM (beats per minute) for the metronome. The TACTL register is set to configure Timer A0 with the ACLK (Auxiliary Clock) as the clock source, and to enable the timer in continuous mode with the MC_2 setting.

Enable Global Interrupts: The next line enables global interrupts by setting the GIE (Global Interrupt Enable) bit in the SR (Status Register) register.

Infinite Loop: Finally, the program enters an infinite loop that continuously waits for interrupts to occur. The microcontroller will execute the interrupt service routines (ISRs) when interrupts are triggered by the button or the timer, allowing the metronome to function based on the BPM and user input.

Interrupt Routines

```
#pragma vector = PORT1_VECTOR
__interrupt void Port1_ISR(void)
{
    if (P1IFG & BIT1)
    {
        P1IFG &= ~BIT1; // Clear interrupt flag for P1.1
        BPM += 10; // Increase BPM by 10
        if (BPM > MAX_BPM) // Check if BPM exceeds maximum limit
        {
            BPM = MAX_BPM; // Limit BPM to maximum value
        }
        TICK_RATE = 60000 / BPM; // Update tick rate based on new BPM
    }
}
```

This is an interrupt service routine (ISR) for Port 1 interrupts. Specifically, it handles interrupts generated by Pin 1.1 (BPM increase button). When Pin 1.1 generates an interrupt (i.e., when the button is pressed), this ISR is executed.
Inside the ISR, P1IFG (Port 1 interrupt flag register) is checked to see if the interrupt was triggered by Pin 1.1 (BIT1). If yes, the interrupt flag for Pin 1.1 is cleared using P1IFG &= ~BIT1. Then, the BPM (beats per minute) global variable is incremented by 10, effectively increasing the BPM by 10 beats per minute. The new BPM is then

checked against the maximum BPM limit (MAX_BPM), and if it exceeds the limit, it is limited to the maximum value (BPM = MAX_BPM). Finally, the TICK_RATE global variable is updated based on the new BPM. Since the metronome tick rate is calculated as 60000 / BPM, this line of code updates the TICK_RATE to reflect the new BPM, which will result in a faster or slower metronome beat depending on whether the BPM was increased or decreased.

```
#pragma vector = PORT2_VECTOR
__interrupt void Port2_ISR(void)
{
    if (P2IFG & BIT1)
    {
        P2IFG &= ~BIT1; // Clear interrupt flag for P2.1
        BPM -= 10; // Decrease BPM by 10
        if (BPM < MIN_BPM) // Check if BPM goes below minimum limit
        {
            BPM = MIN_BPM; // Limit BPM to minimum value
        }
        TICK_RATE = 60000 / BPM; // Update tick rate based on new BPM
    }
}
```

This is another ISR for Port 2 interrupts, specifically for handling interrupts generated by Pin 2.1 (BPM decrease button). The logic is similar to the Port 1 ISR, but in this case, the BPM is decreased by 10 when the Pin 2.1 interrupt is triggered (i.e., when the decrease button is pressed). The BPM is then checked against the minimum BPM limit (MIN_BPM), and if it goes below the limit, it is limited to the minimum value (BPM = MIN_BPM). Finally, the TICK_RATE global variable is updated based on the new BPM.

Code Summary

In summary, this code sets up a metronome using an MSP430F5529 microcontroller. It uses two buttons connected to Pin 1.1 and Pin 2.1 to increase and decrease the BPM of

the metronome, respectively. The BPM is stored in a global variable, and the metronome tick rate is calculated based on the BPM. The tick rate is then used to generate two different tones using Pulse Width Modulation (PWM) with different duty cycles to create the audible beat of the metronome. LED pins are used to provide visual feedback for the beat, and interrupts are used to handle button presses for changing the BPM in real-time.

Note: Please note that the exact implementation and pin assignments may vary depending on the specific hardware and microcontroller used, as well as any additional functionalities or requirements of the metronome. This code serves as a basic example and may need to be modified or adapted to suit your specific needs. Always refer to the microcontroller's datasheet and reference manual for accurate pin configurations, interrupt handling, and other relevant information. Additionally, proper hardware interfacing, debouncing, and other considerations should be taken into account for a robust and reliable metronome implementation.

The full code as a whole can be found in the appendix of this paper, linked here.

Results

The customizable metronome using the MSP430 microcontroller performed as expected and met the project's objectives. The device successfully generated audible beats at the desired tempo and time signature settings, allowing for precise timing during music practice or performance.

To evaluate the performance of the metronome, several tests were conducted. The tempo settings were varied from 30 to 240 beats per minute (BPM), and the time signatures were set to common values such as 4/4, 3/4, and 6/8. The beats generated by the metronome were compared to the expected beats based on the calculated tempo and time signature.

The results obtained showed that the metronome accurately generated the desired beats at different tempo and time signature settings. The generated beats were consistent with the theoretical calculations based on the input settings, indicating a high degree of accuracy in tempo and time signature control. However, during the testing process, some minor issues were encountered. Peak tempos would tend to have latency, as the MSP430 struggled to keep up with the high tempo.

Overall, the results obtained demonstrated that the customizable metronome using the MSP430 microcontroller performed according to expectations and provided accurate and reliable timing control for music practice or performance. The comparison between theory and results confirmed the validity of the theoretical calculations and the functionality of the metronome apparatus.

Discussion

The development of the customizable metronome using the MSP430 microcontroller was largely successful, with the device meeting the project's objectives and generating accurate beats at the desired tempo and time signature settings. However, there are some aspects that went well and some areas that could have been improved.

One of the strengths of the project was the successful implementation of the MSP430 microcontroller as the core of the metronome, allowing for precise control of tempo and time signature settings. The integration of the various hardware components, such as the display, buttons, and speaker, was also successful, resulting in a functional and user-friendly device.

The comparison between the generated beats and the expected beats based on the calculated tempo and time signature showed a high degree of accuracy, indicating that the metronome provided reliable timing control for music practice or performance. The use of graphs to visually represent the data further enhanced the clarity of the results.
However, there are areas that could have been better. The limitations of the MSP430's accuracy and stability occasionally resulted in slight variations in the actual tempo, which could be further improved with the use of more efficient code, or perhaps a buffer for the coming beats of the metronome.

Possible improvements to the device could include the addition of advanced features such as multiple user profiles, rhythmic patterns, or an OLED screen to implement visual BPM and time signature readings. The user interface could be further improved for ease of use, and the device could be made more portable and compact for convenience in different musical settings.

In conclusion, while the customizable metronome using the MSP430 microcontroller achieved its objectives and performed well, there are areas that could be further

improved. The project demonstrated the potential for accurate and reliable timing control in music practice or performance, and future developments could further enhance its functionality and user experience.

Conclusions

The construction of the customizable metronome using the MSP430 microcontroller was a worthwhile endeavour, as it successfully met the project's objectives and demonstrated the potential for accurate and reliable timing control in music practice or performance. Through this project, several important lessons were learned.

First, the practical application of microcontrollers in creating a customizable metronome provided valuable insights into the principles of embedded systems, including hardware integration, software development, and design. The project required understanding of programming concepts, electronic circuitry, and mechanical components, which deepened the understanding of physics and engineering principles.

Second, the project highlighted the importance of efficiency and simplicity with circuits, as well as the power of the MSP430. The initial circuit prototypes were far more advanced, but contained more noise and latency. The final design leveraged the vast capabilities of the MSP430 microcontroller, utilising a simplistic circuit to achieve the desired functionality.

Third, the challenges encountered during the project, such as the limitations of the MSP430’s accuracy and stability, provided valuable experience in troubleshooting and problem-solving in a real-world engineering context.

In conclusion, the construction of the customizable metronome using the MSP430 microcontroller was worth the effort, as it not only achieved its objectives but also provided valuable learning opportunities in embedded systems, testing, and problem-solving. The project contributed to a deeper understanding of the principles behind microcontrollers, and the skills and knowledge gained can be applied in future projects or real-world applications.

To view a short demo of the metronome, please click here (YouTube link).

References

Texas Instruments. (2017). MSP430x2xx Family User's Guide. Retrieved from

https://www.ti.com/lit/ug/slau144j/slau144j.pdf

Appendix

Full C Code

```
#include <msp430f5529.h>

#define BEEP_PIN BIT2   // Pin for beep output
#define LED1_PIN BIT0   // Pin for LED1 output
#define LED2_PIN BIT7   // Pin for LED2 output
#define BEATS 4         // Number of beats
volatile unsigned int BPM = 120; // Global variable for BPM
#define MIN_BPM 30      // Minimum BPM
#define MAX_BPM 240     // Maximum BPM
#define TICK_RATE 60000 / BPM   // Metronome tick rate in milliseconds
#define PWM_FREQUENCY1 2093.005 // Frequency of Tone 1 in Hz (c)
#define PWM_FREQUENCY2 1567.98 // Frequency of Tone 2 in Hz (g)
#define DUTY_CYCLE1 500     // Duty cycle of Tone 1 in tenths of percent
(0 to 1000)
#define DUTY_CYCLE2 250    // Duty cycle of Tone 2 in tenths of percent
(0 to 1000)

void delay_ms(unsigned int ms)
{
    unsigned int i;
    for (i = 0; i < ms; i++)
    {
        __delay_cycles(1000);   // Delay 1000 cycles at 1MHz clock
    }
}

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;   // Stop watchdog timer

    P1DIR |= BEEP_PIN;   // Set LED pin as output
    P1DIR |= LED1_PIN;   // Set LED pin as output
    P1OUT &= ~LED1_PIN;   // Initialize LED pin to low
    P4DIR |= LED2_PIN;   // Set LED pin as output
    P4OUT &= ~LED2_PIN;   // Initialize LED pin to low

    P1DIR &= ~BIT1;     // Set P1.1 as input
    P1REN |= BIT1;      // Enable pull-up resistor on P1.1
```

```
    P1OUT |= BIT1;      // Set pull-up resistor on P1.1
    P1IES |= BIT1;      // Set P1.1 to trigger on falling edge
    P1IE |= BIT1;       // Enable interrupt for P1.1

    TA0CCR0 = 1000 - 1;                  // PWM Period
    TA0CCTL1 = OUTMOD_7;                 // CCR1 reset/set
    TA0CCR1 = (DUTY_CYCLE1 * TA0CCR0) / 1000; // CCR1 PWM duty cycle
    TA0CCTL2 = OUTMOD_7;                 // CCR2 reset/set
    TA0CCR2 = (DUTY_CYCLE2 * TA0CCR0) / 1000; // CCR2 PWM duty cycle
    TA0CTL = TASSEL_2 + MC_1 + TACLR;    // SMCLK, Up mode, Clear TAR

    P2DIR &= ~BIT1;     // Set P2.1 as input
    P2REN |= BIT1;      // Enable pull-up resistor on P2.1
    P2OUT |= BIT1;      // Set pull-up resistor on P2.1
    P2IES |= BIT1;      // Set P2.1 to trigger on falling edge
    P2IE |= BIT1;       // Enable interrupt for P2.1

    __bis_SR_register(GIE); // Enable global interrupts

    while (1)
    {
        P1OUT ^= LED1_PIN;   // Toggle LED pin
        P1SEL |= BIT2; // Pin 1.2 selected as PWM
        TA0CCR0 = 1000000 / PWM_FREQUENCY1 - 1; // Update PWM period for
Tone 1
        TA0CCR1 = (DUTY_CYCLE1 * TA0CCR0) / 1000; // Update PWM duty
cycle for Tone 1
        delay_ms(50);   // Delay for tick rate
        P1SEL &= ~BIT2; // Pin 1.2 selected as PWM
        P1OUT ^= LED1_PIN;   // Toggle LED pin

     //   if ((BPM >= MAX_BPM) || (BPM <= MIN_BPM))
     //   {
     //       P6OUT &= LIMLED;   // Make LED high
     //   }

        delay_ms(TICK_RATE);   // Delay for tick rate

        int i;
        for (i = 1; i < BEATS; ++i)
        {
            P4OUT ^= LED2_PIN;   // Toggle LED pin
            //P1OUT ^= BEEP_PIN;   // Toggle BEEP pin
            P1SEL |= BIT2; // Pin 1.2 selected as PWM
            TA0CCR0 = 1000000 / PWM_FREQUENCY2 - 1; // Update PWM period
for Tone 2
```

```
            TA0CCR2 = (DUTY_CYCLE2 * TA0CCR0) / 1000; // Update PWM duty
cycle for Tone 2
            delay_ms(50);   // Delay for tick rate
            P1SEL &= ~BIT2; // Pin 1.2 selected as PWM
            P4OUT ^= LED2_PIN;   // Toggle LED pin

            delay_ms(TICK_RATE);   // Delay for tick rate
        }
    }

    return 0;
}

#pragma vector = PORT1_VECTOR
__interrupt void Port1_ISR(void)
{
    if (P1IFG & BIT1)
    { // Check if P1.1 triggered
        if (BPM < MAX_BPM)
        { // Check if BPM is less than maximum BPM
            BPM++; // Increase BPM by 1
        }
    }
    P1IFG &= ~BIT1; // Clear P1.1 interrupt flag
}

#pragma vector = PORT2_VECTOR
__interrupt void Port2_ISR(void)
{
    if (P2IFG & BIT1)
    { // Check if P2.1 triggered
        if (BPM > MIN_BPM)
        { // Check if BPM is greater than minimum BPM
            BPM--; // Decrease BPM by 1
        }
    }
    P2IFG &= ~BIT1; // Clear P2.1 interrupt flag
}
```