Introduction

In this project Several melodies are saved in the microcontroller's memory and with each press on the push button the next melody will be played and so on in just a circle

It’s easy to change the saved melodies in the MCU with your own, using the converter program, the melody from the RTTL format is converted into an array of values that need to be written to the melodies.c file and the project will be compiled again, the converter program is given in the repository.

The microcontroller is clocked from an External oscillator with a frequency of 8 MHz. Instead of using ATtiny45, you used the ATmega32 microcontroller.

Requirements

ATMEGA32A

1 × 185$^{.00}$ EGP

Active Buzzer 5V

1 × 6$^{.50}$ EGP

Push button (6x6x6 mm Tack Switch Standard type)

1 × 0$^{.75}$ EGP

White Cap For 4 Pin Push Button

1 × 1$^{.50}$ EGP

Subtotal: 193$^{.75}$ EGP

External Interrupt

The External Interrupts are triggered by INT0, INT1, INT2 pins. Observe that, if it was enabled then the interrupt controller will trigger the configured event if they are configured outputs.

The external interrupts can be triggered by a falling or rising edge or a low level This is set up as indicated in the specification for the MCU Control Register – MCUCR – and MCU Control and Status Register – MCUCSR.

The MCU Control Register contains control bits for interrupt sense control and general MCU functions.

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 35. Interrupt 0 Sense Control

ISC01	ISC00	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

In this project we will use External Interrupt 0 and this is activated by the pin INT0 in the Rising Edge as the push button will be connected as Pullup configuration and once it pressed it will be connected to the ground, so the pin will have high logic then once pressed it will go low level and we want it to change the melody once the button is released so we will get interrupt on the rising edge as the button we go its normal case in high level.

Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

When the INT0 bit is set and the I-bit in the Status Register (SREG) is set, the EXTI is enabled.

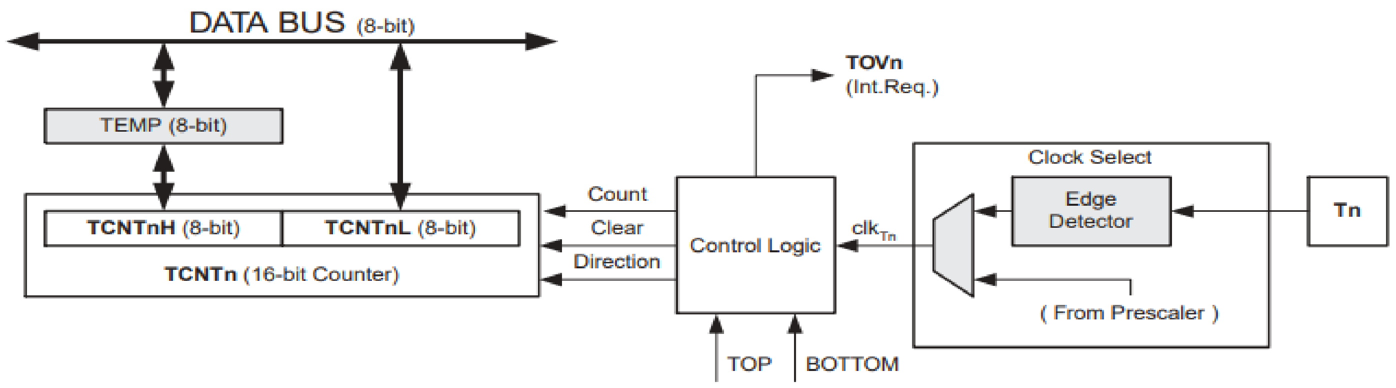
Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

When an edge or logic change on the INT0 pin this triggers an interrupt request, INTF0 becomes set. If the I-bit in SREG and the INT0 bit in GICR are set, the MCU will jump to the corresponding interrupt vector

Timer 1

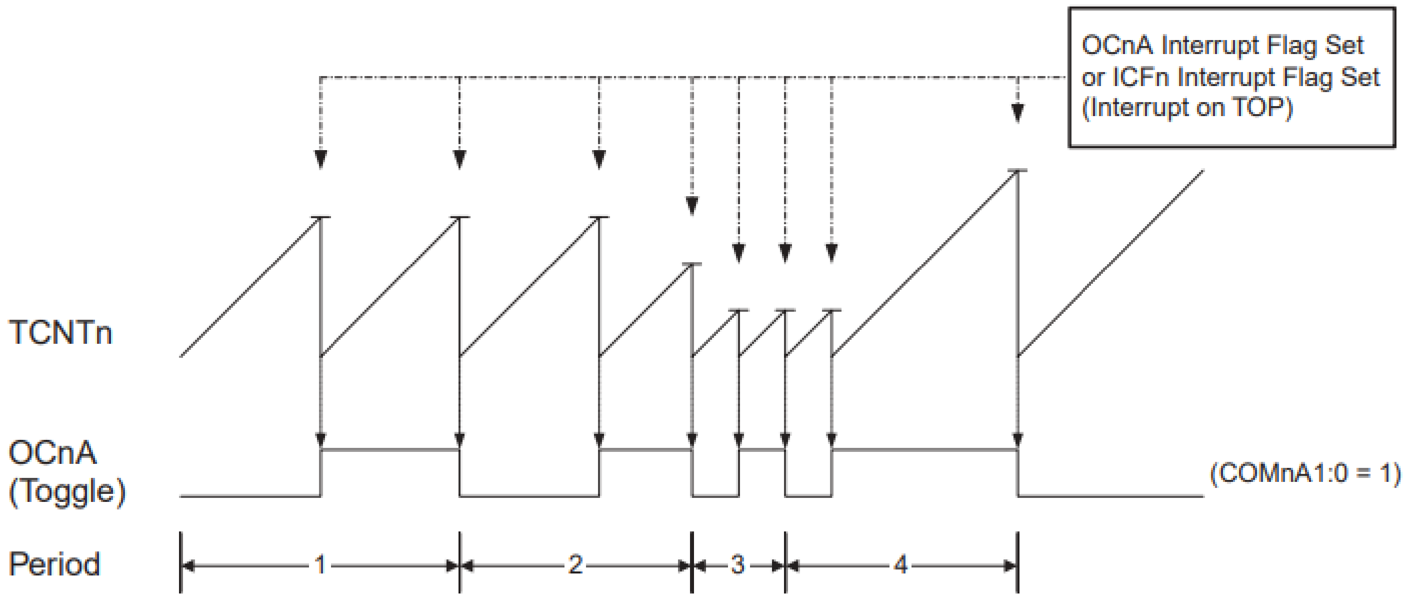
This module in the ATmega32 MCU is 16-bit Timer/Counter in which the TCNT register counts from 0 up to 65535 in which the register will overflow then count back from 0 and Depending on the mode of operation used, the counter is cleared, incremented, or decremented at each timer clock (clkT1).

Figure 41. Counter Unit Block Diagram

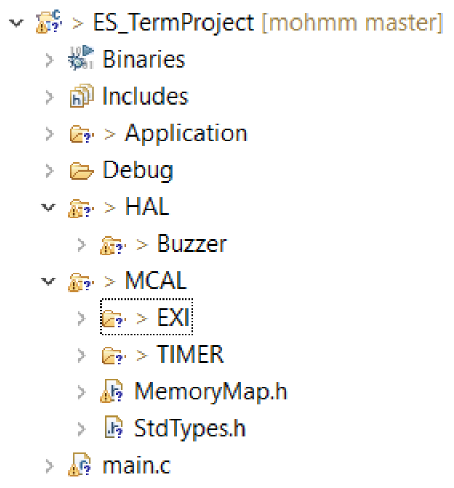


In this project we will deal with the Clear Timer on Compare or CTC mode, the OCR1A or ICR1 Register are used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT1) matches either the OCR1A or the ICR1. The OCR1A or ICR1 define the top value for the counter, hence also its resolution. This mode allows greater control of the compare match output frequency. It also simplifies the operation of counting external events.

Figure 45. CTC Mode, Timing Diagram



Implementation



```
ES_TermProject [mohmm master]
  Binaries
  Includes
  Application
  Debug
  HAL
    Buzzer
  MCAL
    EXI
    TIMER
    MemoryMap.h
    StdTypes.h
  main.c
```

First in our file structure we will work based on the layered architecture system in which we have three main layers :

- Application which contains the main logic or algorithm of the project.
- HAL which contains the External Hardware drivers Abstraction like the buzzer and the melodies.
- MCAL which contains the Microcontroller drivers Abstraction like External Interrupt and Timer.

Main.c

It contains the main logic of the project as first we initialize the system peripherals like IO Pins, Timer, and External Interrupt.

Then we read the button state as it configured in Pullup so Initially and by default it will have high logic and by pressing the button it will go low level and once released it will go high again.

Then we check if the button is low, it means that the button is pressed, and we will play the melody which have the index gActMelody otherwise no melody will be played.

```c
#include "Application/app.h"

volatile uint8 gActMelody = 0;

void main(void)
{
    System_Initialize();
    uint8_t ButtonState = 0;
    while (1)
    {
        ButtonState = READ_BIT(PIND, PIND2);

        if(ButtonState == 0)
            PlayMelody(gActMelody);

        else
            TMR1_SET_OCR1A_VALUE(0);
    }
}
```

The variable gActMelody is configured to be incremented in the ISR related to EXI0 on the rising edge this means that it will be incremented once pressed and released again.

App.c

First we will configure the external interrupt to work on EXI0 to trigger the Rising Edge

```c
ST_externalInterrupt_t exi_0 =
{
        .source = EXI_INT0,
        .edge = EXI_INT0_GENERATE_INTERRUPT_AT_RISING_EDGE
};
```

then we will configure the IO Pins as :

the Push Button Pin (PIND5) will be Input and configured as Pullup to reduce the hardware complexity.

then the Buzzer Pin (PIND2) will be configured as output.

```
void ConfugureIOPins()
{
    // set PIND5 as output for the buzzer
    DDRD |= (1<<PIND5);

    // set PINB2 as input for the push button
    CLEAR_BIT(DDRD, 2);
    SET_BIT(PORTD, 2);
}
```

This function is being called from the main to configure the system so first it will open the global interrupt then configure the Timer, IO Pins and the External Interrupt.

```
void System_Initialize()
{
    sei();
    ConfigureTimer();
    ConfugureIOPins();
    EXI_interrupt_init(&exi_0);
}
```

Timer.h

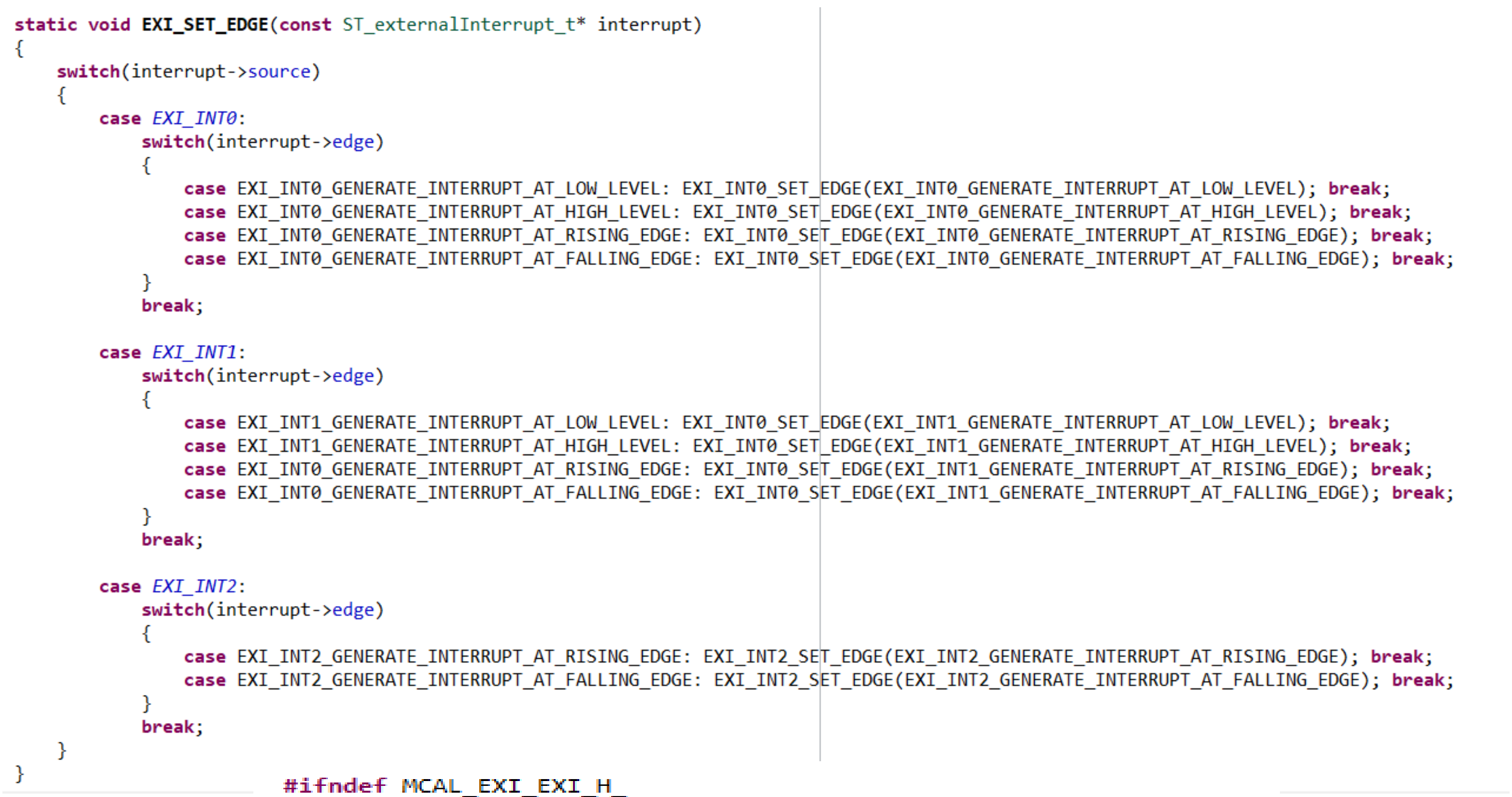
```
#ifndef MCAL_TIMER_TIMER_H_
#define MCAL_TIMER_TIMER_H_

#include "../StdTypes.h"
#include "../MemoryMap.h"

#define TMR1_CTC_OCR1A_MODE      1

#define TMR1_NO_CLOCK_SOURCE    0X00
#define TMR1_1_PRESCALER        0X01
#define TMR1_8_PRESCALER        0X02
#define TMR1_64_PRESCALER       0X03
#define TMR1_256_PRESCALER      0X04
#define TMR1_1024_PRESCALER     0X05

#define TMR1_SET_TCNT1_VALUE(_VAL)       (TCNT1 = _VAL)
#define TMR1_SET_OCR1A_VALUE(_VAL)       (OCR1A = _VAL)
#define TMR1_SET_MOOD(_MOOD)             (TCCR1B |= (_MOOD<<3))
#define TMR1_SET_PRESCALER(_PRESCALER)   (TCCR1B |= _PRESCALER)
#define TMR1_SET_COMPARE_MATCH(_COM)     (TCCR1A |= (_COM<<6))

void ConfigureTimer();

#endif /* MCAL_TIMER_TIMER_H_ */
```

Timer.c

```c
#include "timer.h"

void ConfigureTimer()
{
    // Clears Timer/Counter Control Register A indicating
    // No special modes or configurations for the OC pins
    TCCR1A = 0x00;

    // configuring the prescaler to divide the clock by 64
    TMR1_SET_PRESCALER(TMR1_64_PRESCALER);

    // configuring the timer mode CTC OCR1A
    TMR1_SET_MOOD(TMR1_CTC_OCR1A_MODE);

    //initializes the Timer/Counter 1 register to 0
    TMR1_SET_TCNT1_VALUE(0);
}
```

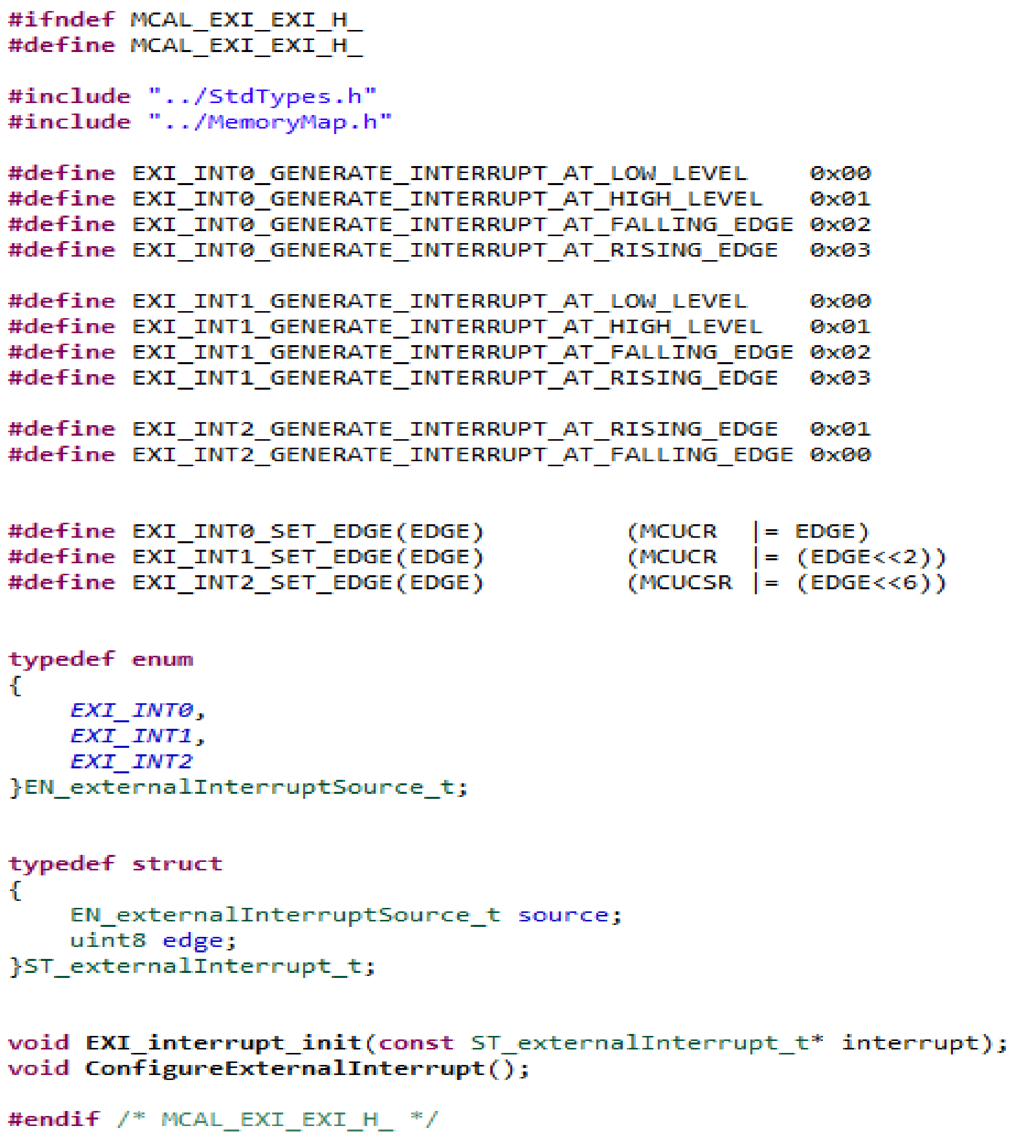
EXI.c

```c
void EXI_interrupt_init(const ST_externalInterrupt_t* interrupt)
{
    if(NULL == interrupt)
        return;

    else
    {
        EXI_SET_SOURCE(interrupt);
        EXI_SET_EDGE(interrupt);
    }
}

static void EXI_SET_SOURCE(const ST_externalInterrupt_t* interrupt)
{
    switch(interrupt->source)
    {
        case EXI_INT0: SET_BIT(GICR,INT0); break;
        case EXI_INT1: SET_BIT(GICR,INT1); break;
        case EXI_INT2: SET_BIT(GICR,INT2); break;
    }
}

ISR(INT0_vect)
{
    gActMelody++;
}
```



```c
static void EXI_SET_EDGE(const ST_externalInterrupt_t* interrupt)
{
    switch(interrupt->source)
    {
        case EXI_INT0:
            switch(interrupt->edge)
            {
                case EXI_INT0_GENERATE_INTERRUPT_AT_LOW_LEVEL: EXI_INT0_SET_EDGE(EXI_INT0_GENERATE_INTERRUPT_AT_LOW_LEVEL); break;
                case EXI_INT0_GENERATE_INTERRUPT_AT_HIGH_LEVEL: EXI_INT0_SET_EDGE(EXI_INT0_GENERATE_INTERRUPT_AT_HIGH_LEVEL); break;
                case EXI_INT0_GENERATE_INTERRUPT_AT_RISING_EDGE: EXI_INT0_SET_EDGE(EXI_INT0_GENERATE_INTERRUPT_AT_RISING_EDGE); break;
                case EXI_INT0_GENERATE_INTERRUPT_AT_FALLING_EDGE: EXI_INT0_SET_EDGE(EXI_INT0_GENERATE_INTERRUPT_AT_FALLING_EDGE); break;
            }
            break;

        case EXI_INT1:
            switch(interrupt->edge)
            {
                case EXI_INT1_GENERATE_INTERRUPT_AT_LOW_LEVEL: EXI_INT0_SET_EDGE(EXI_INT1_GENERATE_INTERRUPT_AT_LOW_LEVEL); break;
                case EXI_INT1_GENERATE_INTERRUPT_AT_HIGH_LEVEL: EXI_INT0_SET_EDGE(EXI_INT1_GENERATE_INTERRUPT_AT_HIGH_LEVEL); break;
                case EXI_INT0_GENERATE_INTERRUPT_AT_RISING_EDGE: EXI_INT0_SET_EDGE(EXI_INT1_GENERATE_INTERRUPT_AT_RISING_EDGE); break;
                case EXI_INT0_GENERATE_INTERRUPT_AT_FALLING_EDGE: EXI_INT0_SET_EDGE(EXI_INT1_GENERATE_INTERRUPT_AT_FALLING_EDGE); break;
            }
            break;

        case EXI_INT2:
            switch(interrupt->edge)
            {
                case EXI_INT2_GENERATE_INTERRUPT_AT_RISING_EDGE: EXI_INT2_SET_EDGE(EXI_INT2_GENERATE_INTERRUPT_AT_RISING_EDGE); break;
                case EXI_INT2_GENERATE_INTERRUPT_AT_FALLING_EDGE: EXI_INT2_SET_EDGE(EXI_INT2_GENERATE_INTERRUPT_AT_FALLING_EDGE); break;
            }
            break;
    }
}
```

EXI.h



```c
#ifndef MCAL_EXI_EXI_H_
#define MCAL_EXI_EXI_H_

#include "../StdTypes.h"
#include "../MemoryMap.h"

#define EXI_INT0_GENERATE_INTERRUPT_AT_LOW_LEVEL    0x00
#define EXI_INT0_GENERATE_INTERRUPT_AT_HIGH_LEVEL   0x01
#define EXI_INT0_GENERATE_INTERRUPT_AT_FALLING_EDGE 0x02
#define EXI_INT0_GENERATE_INTERRUPT_AT_RISING_EDGE  0x03

#define EXI_INT1_GENERATE_INTERRUPT_AT_LOW_LEVEL    0x00
#define EXI_INT1_GENERATE_INTERRUPT_AT_HIGH_LEVEL   0x01
#define EXI_INT1_GENERATE_INTERRUPT_AT_FALLING_EDGE 0x02
#define EXI_INT1_GENERATE_INTERRUPT_AT_RISING_EDGE  0x03

#define EXI_INT2_GENERATE_INTERRUPT_AT_RISING_EDGE  0x01
#define EXI_INT2_GENERATE_INTERRUPT_AT_FALLING_EDGE 0x00


#define EXI_INT0_SET_EDGE(EDGE)         (MCUCR  |= EDGE)
#define EXI_INT1_SET_EDGE(EDGE)         (MCUCR  |= (EDGE<<2))
#define EXI_INT2_SET_EDGE(EDGE)         (MCUCSR |= (EDGE<<6))


typedef enum
{
    EXI_INT0,
    EXI_INT1,
    EXI_INT2
}EN_externalInterruptSource_t;


typedef struct
{
    EN_externalInterruptSource_t source;
    uint8 edge;
}ST_externalInterrupt_t;


void EXI_interrupt_init(const ST_externalInterrupt_t* interrupt);
void ConfigureExternalInterrupt();

#endif /* MCAL_EXI_EXI_H_ */
```

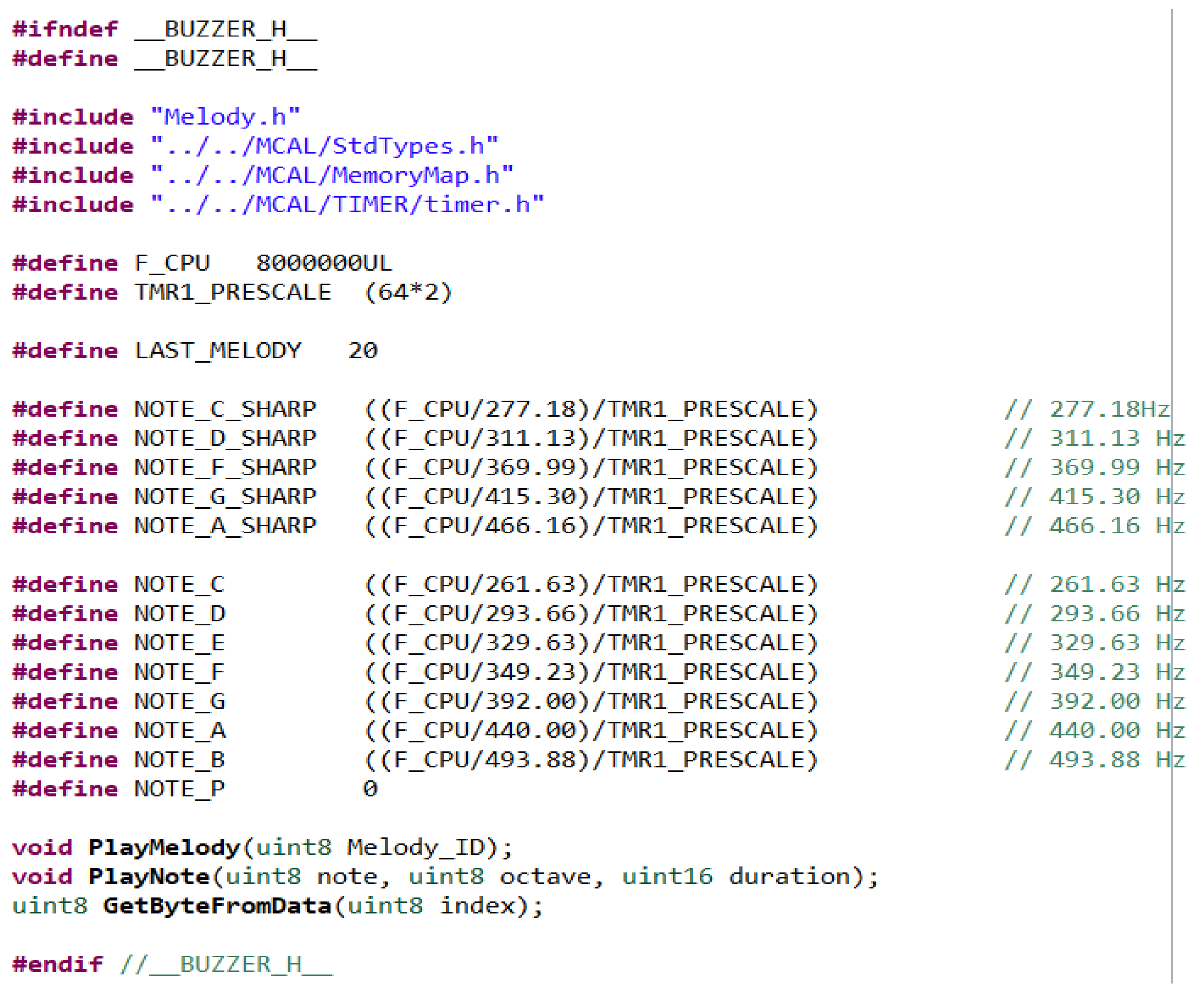
Melody.c

```c
#include "Melody.h"


const unsigned char    Melody001[54]  =  {31,8,172,8,169,4,165,8,165,8,165,4,165,8,165,8,165,8,165,8,165,8,164,8,165,4,167
const unsigned char    Melody002[42]  =  {40,4,196,8,171,2,196,16,160,8,169,4,171,132,164,8,160,16,168,16,171,8,196,8,171,
const unsigned char    Melody003[76]  =  {80,4,161,8,163,8,163,4,163,2,163,4,161,4,161,4,161,4,161,8,164,8,164,4,164,2,164
const unsigned char    Melody004[48]  =  {40,132,193,4,197,4,199,8,202,132,200,4,197,4,193,8,170,8,167,8,167,8,167,2,168,8
const unsigned char    Melody005[96]  =  {31,136,195,16,197,8,200,8,200,16,197,16,195,16,170,16,172,16,195,16,172,8,197,16
const unsigned char    Melody006[118] =  {50,4,198,8,197,4,195,8,194,4,193,8,172,4,171,8,170,8,168,8,170,8,171,4,170,8,168
const unsigned char    Melody007[70]  =  {31,8,165,132,170,8,165,132,197,8,170,4,194,8,172,8,194,4,170,4,165,4,170,4,199,4
const unsigned char    Melody008[54]  =  {35,4,193,8,198,8,197,8,195,8,193,132,170,8,193,8,198,8,197,8,195,8,196,132,197,8
const unsigned char    Melody009[112] =  {62,4,165,8,160,8,166,8,168,8,160,1,193,136,160,4,163,8,160,8,165,1,166,132,160,4
const unsigned char    Melody010[52]  =  {31,8,193,8,193,8,171,8,193,8,128,8,168,8,128,8,168,8,193,8,198,8,197,8,193,2,128
const unsigned char    Melody011[56]  =  {40,8,161,4,166,8,170,4,166,8,161,4,140,2,168,8,166,4,165,8,168,4,165,8,133,4,138
const unsigned char    Melody012[72]  =  {40,8,170,8,193,8,193,8,193,4,171,8,170,4,193,2,192,8,193,8,195,8,171,8,195,4,198
const unsigned char    Melody013[64]  =  {40,8,193,8,171,8,193,8,168,8,164,8,168,4,161,8,193,8,171,8,193,8,168,8,164,8,168
const unsigned char    Melody014[18]  =  {16,136,166,16,171,132,196,8,195,144,171,144,168,144,193,4,198,0};
const unsigned char    Melody015[68]  =  {80,4,195,4,200,4,168,4,200,8,195,8,197,8,195,8,172,4,168,4,163,8,168,8,170,8,172
const unsigned char    Melody016[106] =  {28,32,140,32,128,32,172,32,128,32,167,32,128,32,164,32,128,32,172,32,167,16,128,
const unsigned char    Melody017[48]  =  {40,4,167,136,170,8,167,16,167,8,172,8,167,8,165,4,167,136,194,8,167,16,167,8,195
const unsigned char    Melody018[62]  =  {15,16,165,32,166,32,165,8,172,16,197,32,198,32,197,8,172,16,165,32,166,32,165,16
const unsigned char    Melody019[64]  =  {25,4,193,4,198,16,193,8,195,8,171,4,168,32,160,4,193,16,170,8,166,8,170,4,168,16
const unsigned char    Melody020[54]  =  {40,8,164,8,165,2,160,8,167,8,168,2,160,8,164,8,165,16,160,8,167,8,168,16,160,8,1
```

Melody.h

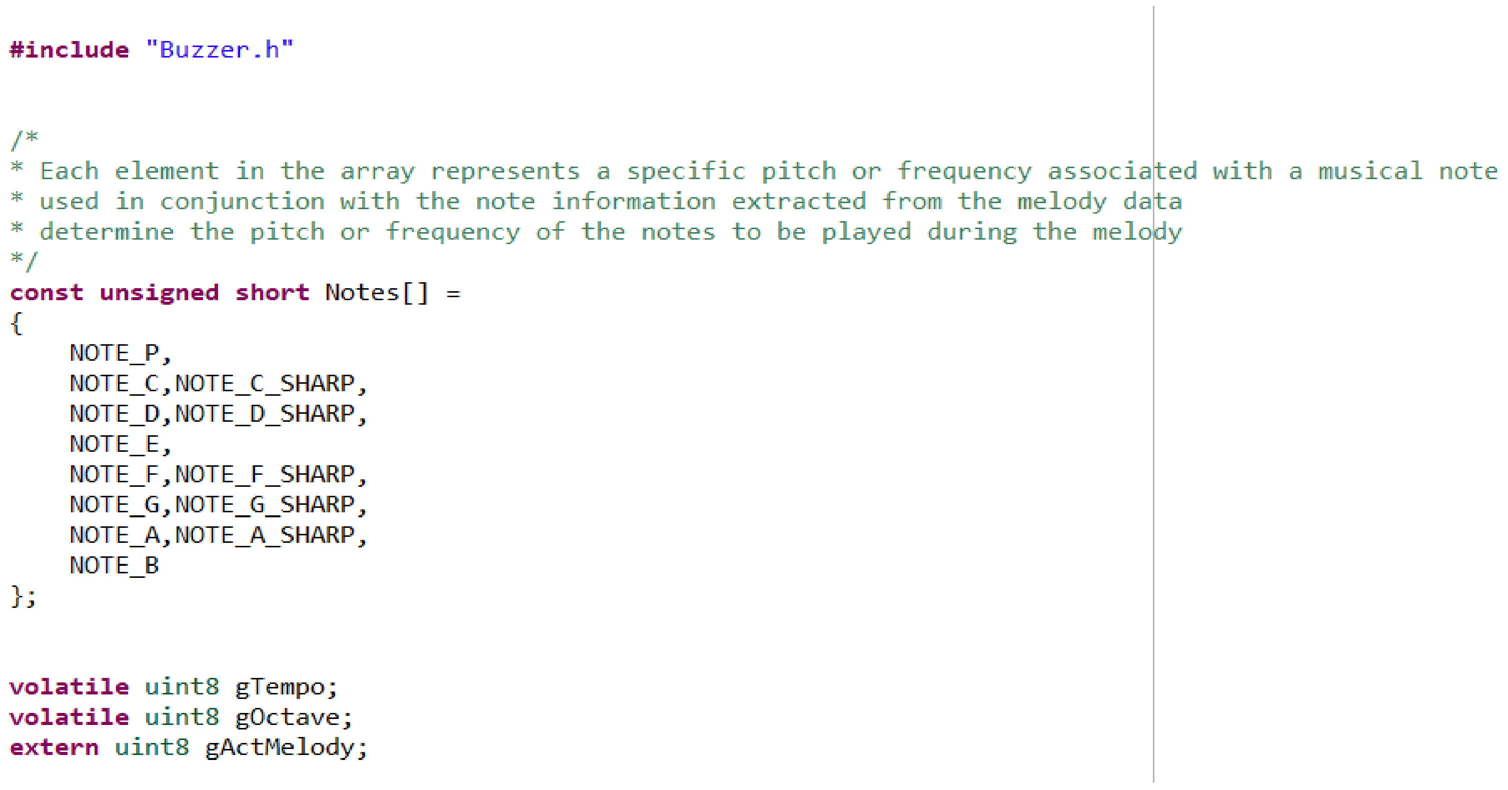
```c
#ifndef __MELORY_H__
#define __MELORY_H__


extern const unsigned char    Melody001[54];
extern const unsigned char    Melody002[42];
extern const unsigned char    Melody003[76];
extern const unsigned char    Melody004[48];
extern const unsigned char    Melody005[96];
extern const unsigned char    Melody006[118];
extern const unsigned char    Melody007[70];
extern const unsigned char    Melody008[54];
extern const unsigned char    Melody009[112];
extern const unsigned char    Melody010[52];
extern const unsigned char    Melody011[56];
extern const unsigned char    Melody012[72];
extern const unsigned char    Melody013[64];
extern const unsigned char    Melody014[18];
extern const unsigned char    Melody015[68];
extern const unsigned char    Melody016[106];
extern const unsigned char    Melody017[48];
extern const unsigned char    Melody018[62];
extern const unsigned char    Melody019[64];
extern const unsigned char    Melody020[54];


#endif //__MELORY_H__
```

Buzzer.h



```c
#ifndef __BUZZER_H__
#define __BUZZER_H__

#include "Melody.h"
#include "../../MCAL/StdTypes.h"
#include "../../MCAL/MemoryMap.h"
#include "../../MCAL/TIMER/timer.h"

#define F_CPU   8000000UL
#define TMR1_PRESCALE  (64*2)

#define LAST_MELODY   20

#define NOTE_C_SHARP   ((F_CPU/277.18)/TMR1_PRESCALE)          // 277.18Hz
#define NOTE_D_SHARP   ((F_CPU/311.13)/TMR1_PRESCALE)          // 311.13 Hz
#define NOTE_F_SHARP   ((F_CPU/369.99)/TMR1_PRESCALE)          // 369.99 Hz
#define NOTE_G_SHARP   ((F_CPU/415.30)/TMR1_PRESCALE)          // 415.30 Hz
#define NOTE_A_SHARP   ((F_CPU/466.16)/TMR1_PRESCALE)          // 466.16 Hz

#define NOTE_C         ((F_CPU/261.63)/TMR1_PRESCALE)          // 261.63 Hz
#define NOTE_D         ((F_CPU/293.66)/TMR1_PRESCALE)          // 293.66 Hz
#define NOTE_E         ((F_CPU/329.63)/TMR1_PRESCALE)          // 329.63 Hz
#define NOTE_F         ((F_CPU/349.23)/TMR1_PRESCALE)          // 349.23 Hz
#define NOTE_G         ((F_CPU/392.00)/TMR1_PRESCALE)          // 392.00 Hz
#define NOTE_A         ((F_CPU/440.00)/TMR1_PRESCALE)          // 440.00 Hz
#define NOTE_B         ((F_CPU/493.88)/TMR1_PRESCALE)          // 493.88 Hz
#define NOTE_P         0

void PlayMelody(uint8 Melody_ID);
void PlayNote(uint8 note, uint8 octave, uint16 duration);
uint8 GetByteFromData(uint8 index);

#endif //__BUZZER_H__
```

Buzzer.c

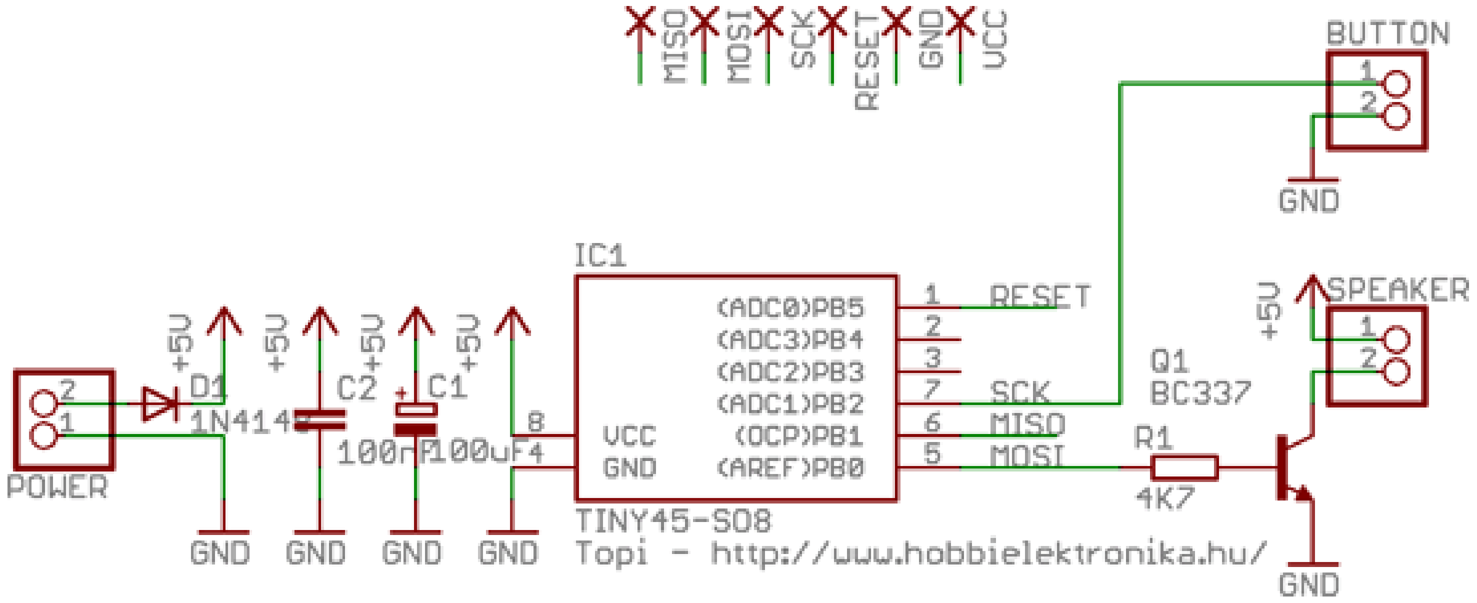


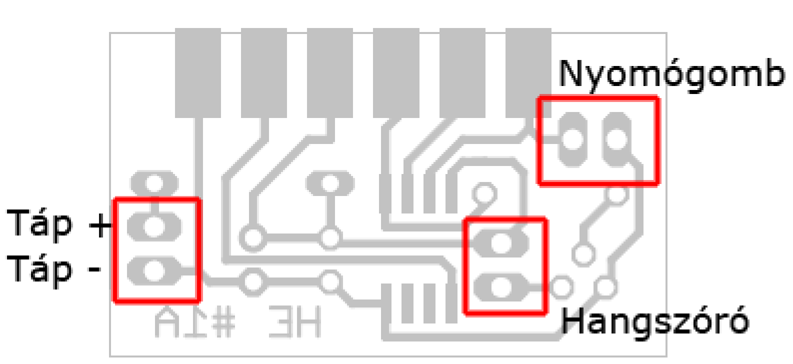
```c
#include "Buzzer.h"


/*
* Each element in the array represents a specific pitch or frequency associated with a musical note
* used in conjunction with the note information extracted from the melody data
* determine the pitch or frequency of the notes to be played during the melody
*/
const unsigned short Notes[] =
{
    NOTE_P,
    NOTE_C,NOTE_C_SHARP,
    NOTE_D,NOTE_D_SHARP,
    NOTE_E,
    NOTE_F,NOTE_F_SHARP,
    NOTE_G,NOTE_G_SHARP,
    NOTE_A,NOTE_A_SHARP,
    NOTE_B
};


volatile uint8 gTempo;
volatile uint8 gOctave;
extern uint8 gActMelody;
```

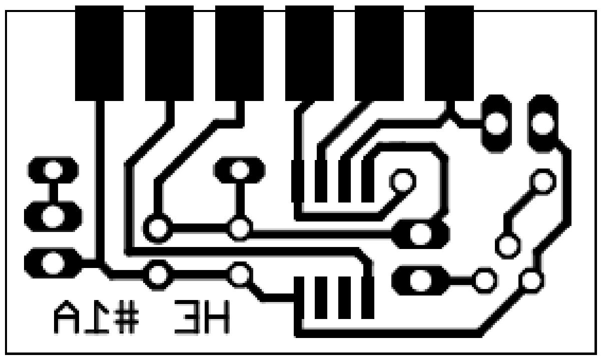
```c
/*
* This function plays the musical note.
* It adjusts the note frequency based on the octave.
* align with the range of octaves in a musical context.
* Octave numbering in music often starts at 1,
* and each increase in octave number represents a doubling of frequency
*/
void PlayNote(uint8 note, uint8 octave, uint16 duration)
{
    // Changing the octave involves doubling or halving the frequency of the note
    // Shifting to a higher octave means doubling the frequency
    // Shifting to a lower octave means halving the frequency.
    switch (octave)
    {
        case 4 : break;
        case 5 : note = note>>1; break;
        case 6 : note = note>>2; break;
        case 7 : note = note>>3; break;
    }


    /*
    * 60000 used to convert the tempo from beats per minute (BPM) to milliseconds per beat
    * gTempo holds the tempo information.
    * 60000 / gTempo calculates the duration of one beat in milliseconds based on the tempo
    * The duration is typically expressed as a fraction of a whole note (e.g., quarter note, eighth note)
    */
    uint16 adur = (60000/gTempo/duration);

    // playing a musical note based on the calculated duration (adur) and frequency (note).
    if (note == 0)
    {
        /*
        * If note is 0, it means the musical note is a rest (no sound),
        * and the program introduces a delay using _delay_ms(adur)
        * This delay represents the duration of the rest.
        */
        _delay_ms(adur);
    }
    // If note is not 0, it means there's an actual musical note to be played
    else
    {
        // resets Timer1's counter to 0.
        TMR1_SET_TCNT1_VALUE(0);

        // sets the Output Compare Register (OCR1A) to the value of note
        // OCR1A value determines the frequency of the waveform generated by Timer1.
        TMR1_SET_OCR1A_VALUE(note);

        // sets timer1 Control Register A (TCCR1A)
        // to enable the Toggle Compare Match (COM1A0) bit.
        TMR1_SET_COMPARE_MATCH(1);

        // introduces a delay of adur milliseconds, representing
        // the duration for which the musical note should be played.
        _delay_ms(adur);

        // After the delay, turn off the PWM signal
        TCCR1A = 0;
    }
}
```

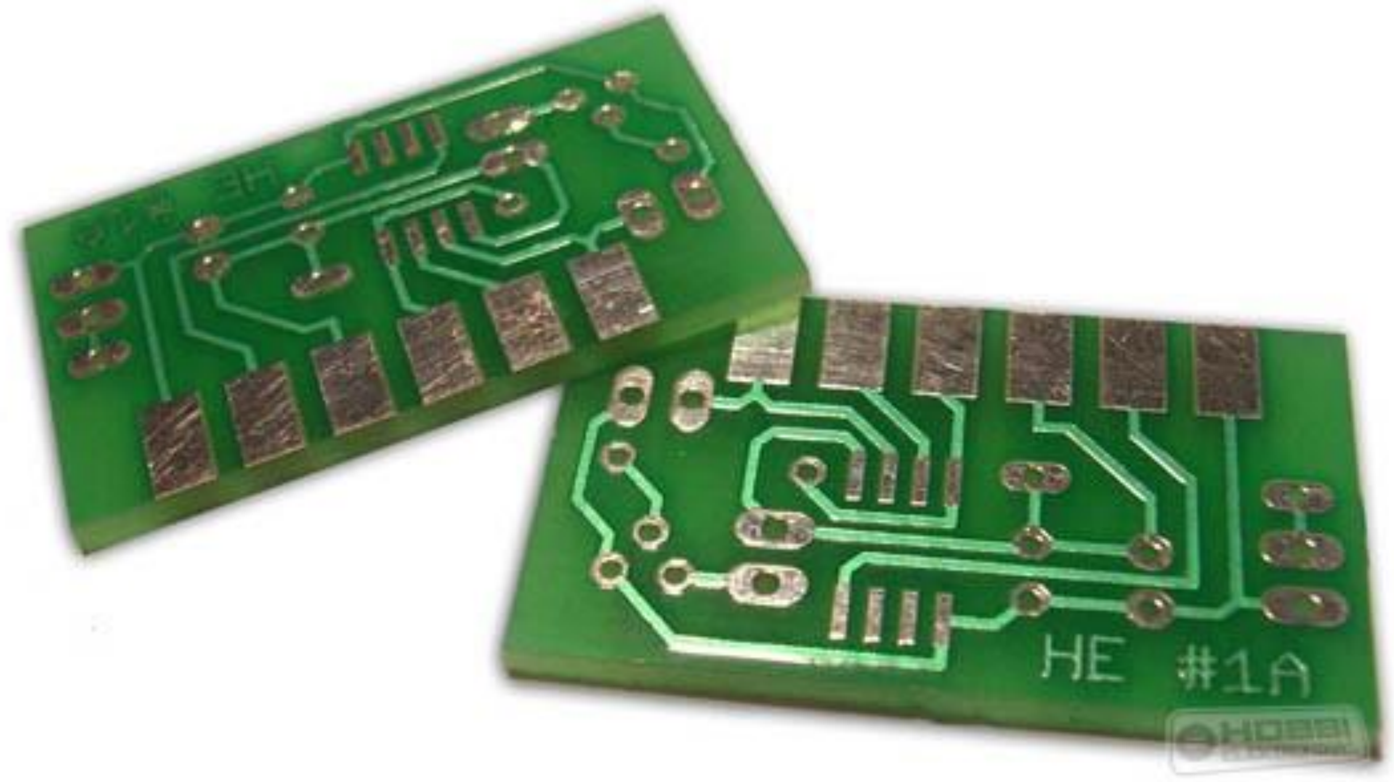
```c
/*
* This function is responsible for playing melody identified by Melody_ID
*/
void PlayMelody(uint8 Melody_ID)
{
    uint8 ChrIndex = 0x00;

    // This holds the duration of the current note
    volatile uint16 duration = 0;

    /*
    * Represents the octave of the current note
    * octave is the interval between one musical pitch and another with half or double its frequency
    * It is set based on the information extracted from the melody data
    */
    volatile uint8 octave = 0;

    /*
    * Represents the frequency of the current note
    * It is set based on the information extracted from the melody data
    * The frequency is used to configure Timer1 for sound generation
    */
    volatile uint16 note = 0;

    /*
    * temp variable to store a byte of data retrieved from the melody
    * It holds information about the note, including its duration,
    * whether it's a rest, the note value, and the octave.
    */
    volatile uint8 ch;

    // hold current melody id
    gActMelody = Melody_ID;

    // Checks if the given Melody_ID is greater than or equal to LAST_MELODY
    if (Melody_ID >= LAST_MELODY)
    {
        // if true it resets the melody, should wrap around to the beginning.
        Melody_ID = 0;

        //Resets the index to the starting position for the melody data.
        ChrIndex = 0x00;
        gActMelody = 0;
    }

    // initializing the gTempo when the melody is just starting
    if(ChrIndex == 0)
    {
        // ChrIndex is incremented to move on to the next piece of data in the melody.
        gTempo = GetByteFromData(ChrIndex++);
    }

    // ch is to store byte of data which retrieved from the melody
    ch = GetByteFromData(ChrIndex++);

    // extracting the duration of a musical note
    duration = ch & 0b00111111;

    // extracting another byte of data from the melody
    ch = GetByteFromData(ChrIndex++);

    // extracts information about the pitch then,
    // access an element from Notes which corresponds to it
    note = pgm_read_byte(Notes+(ch & 0b00001111));

    // extracts the octave information from the byte of data from melody
    octave = (ch & 0b11100000)>>5 ;

    PlayNote(note, octave, duration);
    return;
}
```

System Design

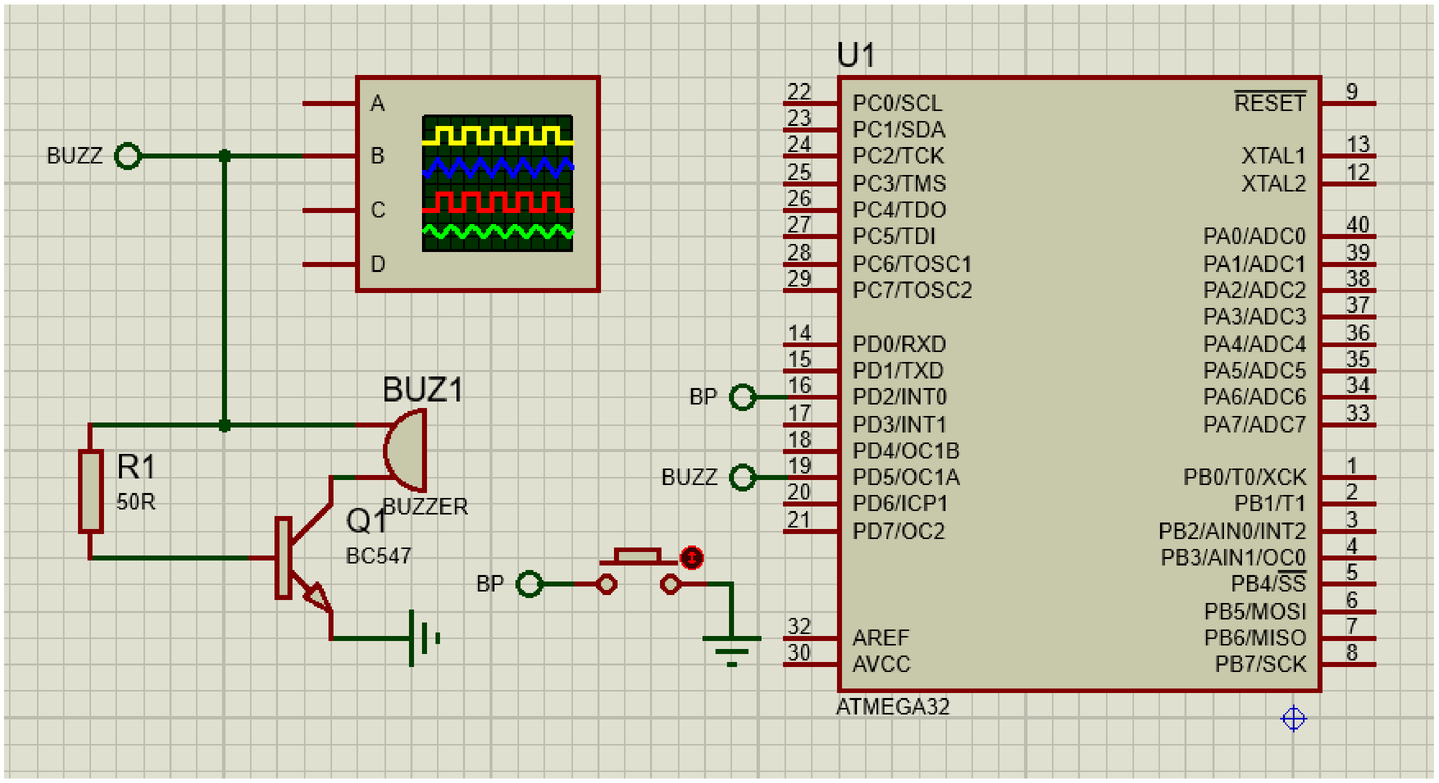








Proteus Design


U1
22
23
24
25
26
27
28
29
14
15
16
17
18
19
20
21
32
30
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B
PD5/OC1A
PD6/ICP1
PD7/OC2
AREF
AVCC
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PB0/T0/XCK
PB1/T1
PB2/AIN0/INT2
PB3/AIN1/OC0
PB4/SS
PB5/MOSI
PB6/MISO
PB7/SCK
9
13
12
40
39
38
37
36
35
34
33
1
2
3
4
5
6
7
8
ATMEGA32
A
B
C
D
BUZZ
BUZ1
BUZZER
R1
50R
Q1
BC547
BP
BUZZ
BP