Guitar tuner
Digitala Projekt (EITF40)

Jakub Gorski, D07 (dt07jg8@student.lth.se)
Patrik Thoresson, D07 (dt07pt2@student.lth.se)

Handledare: Bertil Lindvall

Inlämnad: 2011-03-01

Sammanfattning

This project focuses on constructing a guitar tuner with an `ATmega16`. The tuner will be able to tune any instrument that can be connected to the tuner using a $3.5mm$ analog jack. The difference between this tuner and other tuners is that it will use the `ATmega16`'s internal *ADC* to measure the frequencies, whereas others usually use *Voltage-Controlled Oscillators*.

Innehåll

1 Inledning

Detta projekt var en del av kursen *Digitala och Analoga Projekt*, `EITF40`, på Lunds Tekniska Högskola. Som projekt konstruerades en stämapparat. Syftet med projektet är att skaffa sig ett klart koncept av det man vill konstruera, samtidigt som man behåller utrymme för eventuell vidareutveckling. I detta fallet kommer vidareutvecklingen att anspela på `ATmega16`:s *ADC*[1] som även i fortsättningen kommer möjliggöra en eventuell förvrängning av insignalen som kommer ge stämapparaten möjligheten att även vara en enklare effektprocessor. Resultatet blev en fungerande kromatisk stämapparat som automatiskt detekterar aktuell sträng, varpå den indikerar dess felmarginal och låter användaren stämma gitarren.

2 Kravspecifikation

Syftet med kravspecifikationen var att i ett tidigt stadium bestämma vilka funktioner stämapparaten bör hantera.

2.1 Ursprunglig kravspecifikation

- Kretsen ska kunna användas som en stämapparat, där 7 lysdioder indikerar huruvida instrumentet är stämt i förhållande till önskad frekvens.
- Instrumentet kommer att anslutas via en 3.5mm mono *TRS*-anslutning.
- När kretsen inte är använd efter 1 minut kommer den att gå in i ett standby läge.
- Ljudet från en gitarr ska kunna förvrängas till olika förbestämda effekter.
- För att kunna skifta mellan effekter används knappar varpå information kommer att visas på LCD displayen.

2.2 Uppfyllda specifikationskrav

Under projektets gång märktes det att kravspecifikationen var för ambitiös. Till en början skulle kretsen fungera som en stämapparat och om tiden räckte till skulle gitarreffekter, samt övriga funktionskrav läggas till. Därför lades fokus på de grundläggande funktionerna som en fungerande stämapparat kräver. Inom dessa krav hamnade bland annat signalförstärkning och lysdioder som indikerar strängens felmarginal. Den slutgiltiga stämapparaten uppfyllde följande krav:

- Kretsen ska kunna användas som en stämapparat, där 7 lysdioder indikerar huruvida instrumentet är stämt i förhållande till den önskade frekvensen.
- Instrumentet kommer att anslutas via on 3.5mm mono *TRS* anslutning.
- Tonens felmarginal skall visualiseras med hjälp av 7 lysdioder.

[1] Analog-to-Digital Converter

3 Konstruktion

I vår slutgiltiga konstruktion använde vi oss av följande komponenter:

- 2x OP-förstärkare (*CA3160E*).[2]
- `ATmega16`, 8-bit Mikrokontroller med 16KiB programmerbart minne.[1]
- Analog to Digital Converter (ADC) som är inbyggd i `ATmega16`.
- 3x $10k$ trimpots.
- $5k$ trimpot.
- $1k$ trimpot.
- $500k$ trimpot.
- 2x $18pF$ kondensatorer.
- 5x $4.7nF$ kondensatorer.
- 1x *898-3-R1K* motstånd.
- 7x LED dioder. Varav 4x orange, 2x röda och 1x grön. Meddelar om strängens nuvarande avvikelse.
- *CMACKD* 8MHz kristall.
- 2x $1k$ motstånd.
- $10\mu H$ spole.

4 Metod

4.1 Testkopplingar

Inledningsvis användes ett testbräde för att testa att OP-förstärkaren fungerade som tänkt. Kopplingen ritades upp och kopplades sedan enligt vår design. En mikrofon användes till en början för att testa förstärkningen. Däremot så krävde en mikrofon en annorlunda krets för att fungera med den förstärkning vi tänkt oss. Målet var att förstärka signalen så att vågen täcker intervallet 0-2.5V på oscilloskopet, vilket bör ge en amplitud på $2.5V$. Denna amplitud på insignalen är eftertraktad då `ATmega16`:s ADC har en intern referensspänning på $2.56V$, vilket resulterar i en 8-bitars upplösning på samplingen då ADC:ns interna förstärkning är avstängd.[1, p. 205]

Med denna lösning syntes endast topparna av ljudvågen på oscilloskopet, vilket också tydde på att enbart den positiva halvan av vågen förstärks. Detta på grund av att OP-förstärkaren var jordad och kunde därmed inte förstärka den negativa delen av insignalen. Genom att införa en spänningsdelare mellan mikrofonen och OP-förstärkaren åtgärdades detta. Därmed förstärktes spänningsdifferensen mellan ingångarna på OP:n, vilket gjorde att utspänningen var $1.25V$ när insignalens spänning var $0V$. Den fullständiga, förstärkta vågen, kunde därefter representeras på oscilloskopet med intervallet $0 - 1.25V$, för dess negativa amplitud samt 1.25-2.5V för den positiva delen av vågen.

4.2 Insignal och förstärkning

Eftersom mikrofonen endast användes i testsyften byttes den i detta stadiet ut till förmån av en $3.5mm$ *TRS*-anslutning där en gitarr kopplades in. I denna delen av projektet påbörjades även försök att sampla signalen i programkod, vilket medförde att en $8MHz$ kristall kopplades in tillsammans med två stycken $18nF$ kondensatorer[3] för att accelerera kristallen vid uppstart av `ATmega`:n.[5]

Spänningsdelaren, som justerar spännings-differensen på ovannämnda OP-förstärkare, reglerades av två stycken variabla resistorer, vars inställning gav den önskade utsignalen. Däremot visade det sig vara svårt att kalibrera spänningsdelaren. Detta medförde att en tongenerator användes för att få så exakt ton som möjligt, varpå OP:ns variabla resistorer justerades med assistans av oscilloskop. Efter kalibreringen av OP-förstärkaren blev det uppenbart att insignalens förstärkning, som då var på $500x$, var otillräcklig. Testerna visade att en ytterligare en förstärkare behövdes för att insignalen till ADC:n skulle ha tillräcklig amplitud för att kunna detektera ljudvågen. Dessutom kunde denna förstärkning inte höjas ytterligare utan att drabbas av internt brus, därför infördes en extra OP-förstärkare som kopplades i serie för att nå en maximal amplitud på $2.5V$.[2]

Sammanfattningsvis, vågen som enligt oscilloskopet endast innehar en positiv amplitud, justeras genom förstärkning av spännings-differensen, till att hela signalen kunde ses inom intervallet 0-$2.5V$ på oscilloskopet. Detta medförde också att eventuell tystnad på ingången till stämapparaten gav en $1.25V$ signal på oscilloskopet om den mäts efter de två förstärkarna.

4.3 Hantering av brus

När förstärkarna var på plats noterades det brus på insignalen till ADC:n, vilket påverkade dess mätvärden. För att förebygga detta infördes lågpassfilter med kondensatorer på $4.7nF$ parallellt mellan samtliga strömingångar på komponenterna, samt en spole före A_{vcc}. Detta för att, enligt `ATmega16`-manualen, uppnå *ADC noise cancellation*.[1, p. 210] Även efter dessa modifikationer syntes brus på oscilloskopet. Bruset låg på $1.25V$ på grund av den förstärkta spänningsdifferensen, varpå ADC:n samplade brus. För att undvika sampling av brus ökades förstäkningen av signalen, medan spännings-differensen minskades.[3]

Detta medförde att ADC:n endast såg dalarna i den inkommande vågformen, vilket löste brusproblemen då dalarna i signalen var mindre bruspåverkade än topparna. Tack vare detta stannade bruset på en högre nivå, medan signalens, renare, förstärkta dalar syntes täckte 0-$2.5V$ intervallet. Men även detta orsakade problem, då det krävde en väldigt sträng tröskel i programkoden för registrerering av perioder, vilket orsakade dåliga mätvärden för frekvensen. Anledningen till problemet var att tidsräkningen i `TCNT1` påverkades av bruset, vilket medförde att den mätte felaktig tidsåtgång och därmed frekvens. Detta löstes genom att byta ut $4.7nF$ kondensatorn efter $3.5mm$ anslutningen i första OP-förstärkaren till en ny kondensator, varpå bruset försvann.

När förstärkarna ansågs vara tillräckligt bra på testbrädet, flyttades komponenterna till slutgiltiga kopplingsplattan och löddes till rätt pinnar på processorn.

5 Mjukvara

När förstäkningen av signalen var på en nivå att ADC:n kunde registrera vågen påbörjades utvecklingen av mjukvaran. I detta skedet stämde mätvärdena både från oscilloskopet och ADC:n överens. För att kunna beräkna antalet perioder behövdes det en effektiv metod av frekvensuppmätning. Detta löstes med hjälp av en timer `TCNT1` samt $62.500KHz$ samplingsfrekvens på ADC:n.

5.1 Timers

Valet av `TCNT1` berodde på att `TCNT0` är en 8-bitars räknare och kunde därför endast räkna till 256, vilket orsakade overflow under samplingen som påverkade frekvensuträkningen. En 16-bitars räknare gav en mer exakt tidsuppmätning, vilket gav bättre mätvärden. `TCNT1` användes till en början med en prescaler på 64, vilket gjorde att den räknade upp varje gång klockan på processorn fullbordat 64 klockcykler. Eftersom `TCNT1` räknade för snabbt och gav upphov till overflow med en prescaler på 64, ändrades denna till 256 för att säkra dess tidsbuffert.

Overflow i `TCNT1` användes till en periodisk nollställning av frekvensuträknings-algoritmen, vilket inträffar automatiskt om en insignal inte påträffas under ungefär en sekund. Mer om detta i stycke 5.3.2.

5.2 *ADC* och sampling

Samplingsfrekvensen för ADC:n var satt till $62.500KHz$, vilket mostvarar en prescaler på 128. Valet av denna prescaler berodde på att en prescaler på 256 ger för låg sampling ifall signalen blir högfrekvent, medan en prescaler på, till exempel, 32 ger dålig precision, vilket medför att icke-existerande perioder, eller brus registreras. I detta skede representerades insignalen till ADC:n på intervallet $0V$-$2.5V$ där spännings-differensen försköts så att endast undersidan av vågen hissades ner.

För att samplingar skall kunna påbörjas automatiskt sattes ADC:n i *free running mode*, dock så var detta väldigt processorkrävande då processorn inväntade varje sampel, vilket orsakade stopp i exekveringen när ADC:ns värde tilldelades till en variabel. För åtgärda detta sattes `ADIE` flaggan till 1 i `ADCSRA`, vilket gjorde att ADC:n genererade ett avbrott för varje fullbordad sampel istället för att avbryta exekveringen.

Detta fungerade bra tills oscilloskopet påpekade brusproblem, vilket även påverkade frekvensuträkningen. Som ett försök att lösa problemet utökades ADC:ns läge till *ADC noise canceller*.[1, p. 210] ADC:ns *free running mode* byttes ut mot läget *single conversion mode*, samt så sattes processorn i viloläge före sampling programmets main-loop i syfte att minska digitalt brus.[1, p. 219] Efter slutförd sampel sattes processorn igång igen genom att sätta `SE`-flaggan i `MCUR` till 1.

Även om denna metod minskade bruset, så minskade den även utrymmet för eventuell felsökning. Detta på grund av att den uppmätta signalen på oscilloskopet klipps av för var gång `ATmega16`:n går in i viloläge, vilket omöjliggör felsökning av insignalen. Även den effektiva samplingsfrekvensen minskas till hastigheten som processorn kan gå in respektive ut ur viloläget, vilket påverkar den digitala vågrepresentationen.

Trots ovannämnda åtgärder var bruset så pass stort att ADC:n inte kunde mäta signalen. Det är även nämnvärt att både oscilloskop såsom `JTAG`:en påverkade ADC:ns mätvärde när dessa var påkopplade. Mer om detta nämns i stycke 4.3.

Sammanfattningsvis, när brusfaktorn bedömdes vara ute ur bilden fortskred arbetet med syftet att optimera koden för frekvensuträkningen så att perioder registreras så korrekt som möjligt.

5.3 Optimering av algoritm

5.3.1 Amplitud-tröskel

Perioderna registrerades genom vänta tills ADC:n påträffar lägsta tillståndet på vågen, varpå den markerar det som en period när vågen börjar stiga igen. För att undvika att brus skall registreras som perioder infördes ett minsta tröskelvärde som `ADCH` måste understiga (endast dalarna i sigalen mäts), vilket ger ett striktare villkor för när en period inträffar och ger bättre mätvärden.

En ytterligare kodoptimering kunde utföras genom att hoppa över de första fåtal perioder när en signal som når över tröskeln registreras. Tack vare detta införs det en variabel som först hoppar över ett konstant antal perioder före den riktiga frekvensmätningen, vilket ger insignalen tid att stabilisera.

5.3.2 Period-tröskel

Vid en beräkning med bristande antal registrerade perioder, inväntar algoritmen ytterligare perioder då användaren spelar på gitarren. Dödtiden mellan sista registrerade period och den tillkommande, när användaren spelar på strängen, räknades som en enda lång period.

Detta åtgärdades med `TCNT1`:s *overflow interrupt* som via en avbrottsrutin, när räknaren gått förbi sitt maxvärde, nollställer algoritmens variabler.[1, p. 114] Dock så hjälper inte detta om det kommer en insignal igen innan räknaren hunnit hamna i avbrottsrutinen. Därför infördes en variabel som håller reda på `TCNT1`:s förra värde, med vars hjälp man kan se när senaste perioden inträffat. På så sätt körs algoritmen om när tidsåtgången mellan två perioder är för stor, vilket tyder på att perioderna hör till två olika tillfällen då användaren spelat på gitarren.

5.4 Omräkning av konstanter

När rätt antal perioder har uppmätts, hämtas `TCNT1` som mäter tiden. För att underlätta uträkning av frekvensen räknades frekvensvärdena om. Varje ton som stämapparaten bör kunna detektera[4], omräknas från Hz till ett `TCNT1`-värde enligt nedanstående formel. Detta på grund av att uträkning av frekvens krävde i vanliga fall en division, som kan undvikas om man vänder på problemet. Därför omräknades konstanterna som definierar toner, där divisionen utförs vid kompilering, istället för att utföra den varje gång.

$$\frac{\frac{prescaler_{TCNT1}}{CPU_{freq}} \cdot n_{periods}}{string_{Hz}} = TCNT1 \tag{1}$$

```
------ 12804,777140335392762577228596646
E      11377,42718446601941747572815534
------ 9950,07722859664607237422771 4034
A      8522,72727272727272727272727273
------ 7456,22459601467235055021 3145633
D      6389,72191930207197382769901 85387
------ 5587,41914666614419434545424 5896
G      4785,11637403021641486320947 32544
------ 4290,94632892341859136162255 61635
B      3796,77628381662076786003563 90734
------ 3319,70409096834676961831375 64675
EH     2842,63189812007277137659187 38629
------ 2365,55970527179877313486999 12565
```

Figur 1: Övergångarna mellan strängar som värde av `TCNT1`

Efter denna konvertering kommer uträkningen motsvara värdet på *TCNT1*. Samtliga divisioner i algoritmen är eliminerade.

5.5 Tonläge

När `TCNT1`:s värde är antaget sker identifiering av strängens tonläge. Identifieringen sker med hjälp av jämförelser av likhet med värdena i Figur 1 som räknades ut enligt ekvation (1). Jämförelserna består av tröskelövergångar mellan toner som räknas ut enligt:

$$\frac{(tone_{adjacent\_LOW} - (tone_{adjacent\_HIGH} - tone_{adjacent\_LOW}))}{2} = TONE\ TRANSITION \quad (2)$$

Ekvationen stämmer förutom för den ljusaste tonövergången, då gäller följande:

$$\frac{(tone_E + (tone_E - tone_A))}{2} = EH\_ETOP\ TRANSITION \quad (3)$$

Med hjälp av dessa konstanta övergångströsklar (även uträknade i Figur 1) kan resultatet visualiseras på `LED`-lamporna. Dock så krävs det mindre trösklar för övergångar mellan `LED`-lamporna. Detta sker i enlighet med:

$$\frac{transition_{adjacent\_HIGH} - transition_{adjacent\_LOW}}{6} = LED\ OFFSET \quad (4)$$

Detta ser till att avvikelser från toner kan representeras visuellt med hjälp av `LED`-lamporna.

6 Resultat

Resultatet är en fungerande stämapparat med möjligheten att visa vilken avvikelse strängen har för tonen man stämmer mot. Den är inte särskilt exakt och

har väldigt stränga tröskelvärden och villkor, vilket gör att man kan behöver spela på en sträng ett antal gånger innan signalen registreras som en godtyckligt stabiliserad våg. Insignalen kan fortfarande registreras fel om man råkar få in en oren ton. Till exempel om strängen man spelar på skorrar, eller om man råkar röra volymkontrollen på instrumentet man spelar på. Detta beror på att det bildas högfrekvent brus när man vrider på en potentiometer, på en gitarr likaså, som ytterligare förstärks med de två existerande OP-förstärkarna som används.

Stämapparaten har många variabla resistorer, varav tre reglerar förstärkningen, medan två reglerar spännings-differensen hos första OP-förstärkaren. Det är mycket som kan gå fel med så pass många trimpots, vilket det har gjort. För att kringgå högfrekventa bruset har bland annat spännings-differensen ökats så att endast den negativa delen av vågen syns på intervallet 0-2.5V vilket gav renare perioder. Mer om hur brusproblemen löstes nämns i 4.3.

7 Slutsats

I efterhand känns det som att vi inte borde använt testbrädet under så pass långt tid som vi faktiskt gjorde. Även om kretsen fungerade på testbrädet så skulle samma krets både se annorlunda ut och kopplas annorlunda vid lödning på kopplingsbrädet, vilket var tidskrävande. Vi skulle även kunnat använda oss av en *Voltage-Controlled Oscillator*, vilket skulle dramatiskt minska kodkomplexiteten, men även förenkla hårdvaruimplementationen. Detta på grund av att en *Voltage-Controlled-Oscillator* skickar en impuls varje gång en period inträffar i en signal, vilket förenklar frekvensuträkning. Dock så skulle detta begränsa eventuellt planerad funktionalitets-utökning. Projektet hade inte varit lika givande om det hade byggts med en *Voltage-Controlled-Oscillator*.

Ständiga ändringar på förstärkarkonfigurationer med 5 variabla resistorer och nära samarbete med oscilloskop tog upp en större del av projektet. Implementationen av mjukvara tog drastiskt mindre tid. Detta har gett oss en bättre förståelse för hur en analog signal bör se ut för att den skall kunna digitaliseras.

Referenser

[1] Datablad ATmega16, [online] Adress: <`http://www.eit.lth.se/fileadmin/eit/courses/edi021/data` [Sista åtkomst 24 Februari 2011]

[2] Datablad OP-Amp CA3160. [online] Adress: <`http://www.eit.lth.se/fileadmin/eit/courses/edi021/datablad/Analog/opamp/ca3160.pdf`> [Sista åtkomst 24 Februari 2011]

[3] USB and PIC Microprocessors 16C745 and 18F2455. [online] Från AlanMacek.com. Adress: <`http://www.alanmacek.com/usb/#Project_Hardware`> [Sista åtkomst 24 Februari 2011]

[4] WA's Encyclopedia of Alternate Guitar Tunings. [online] Adress: <`http://members.cox.net/waguitartunings/tunings.htm`> [Sista åtkomst 24 Februari 2011]

[5] Reference Oscillator Crystal Requirements for the MC1320x, MC1321x, MC1322x, and MC1323x IEEE 802.15.4 Devices. [online] Freescale Semiconductor. Adress: <http://cache.freescale.com/files/rf_if/doc/app_note/AN3251.pdf> [Sista åtkomst 24 Februari 2011]

8 Appendix A - Källkod

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

// LED 0-6 DEFINITIONS
#define led0 0b00000001
#define led1 0b00000010
#define led2 0b00000100
#define led3 0b00001000
#define led4 0b00010000
#define led5 0b00100000
#define led6 0b01000000

#define SIGNAL_CUTOFF_THRESHOLD 0x97
#define SAMPLES 37
#define BURN 10       // how many edges to skip to allow signal to
    stabilize
#define BURN_RESULT 3 // results to skip once a string has been
    detected


#define T1_PRESCALER 256
#define FREQUENCY         8000000                // CPU CLOCK IN Hz
#define BASE_FREQUENCY  (FREQUENCY/T1_PRESCALER) // TIMER/COUNTER1
    COUNTS PER SECOND

// ACTUAL FREQUENCIES
#define ACOUSTIC_E  82.4        // low string (1)
#define ACOUSTIC_A  110
#define ACOUSTIC_D  146.72
#define ACOUSTIC_G  195.92
#define ACOUSTIC_B  246.92
#define ACOUSTIC_EH 329.8       // high string (6)

/*
STRING OFFSET CALCULATION TABLE
      KEY:
      "STRING" = OFFSET_STRING
      "------" = TRANSITION

      //------12804,777140335392762577228596646
      // E  11377,427184466019417475728155 34
      //------9950,077228596646072374227714034
      // A  8522,72727272727272727272727273
```

```
      //------7456,22459601467235055021314563 3
      // D  6389,72191930207197382769901853 87
      //------5587,41914666614419434545424589 6
      // G  4785,11637403021641486320947325 44
      //------4290,94632892341859136162255616 35
      // B  3796,77628381662076786003563907 34
      //------3319,70409096834676961831375646 75
      // EH 2842,63189812007277137659187386 29
      //------2365,55970527179877313486999125 65
*/

// FREQUENCIES AS COUNTER VALUES
#define OFFSET_E    ((BASE_FREQUENCY*SAMPLES)/ACOUSTIC_E)     // Low E
#define OFFSET_A    ((BASE_FREQUENCY*SAMPLES)/ACOUSTIC_A)     // A
#define OFFSET_D    ((BASE_FREQUENCY*SAMPLES)/ACOUSTIC_D)     // D
#define OFFSET_G    ((BASE_FREQUENCY*SAMPLES)/ACOUSTIC_G)     // G
#define OFFSET_B    ((BASE_FREQUENCY*SAMPLES)/ACOUSTIC_B)     // B
#define OFFSET_EH   ((BASE_FREQUENCY*SAMPLES)/ACOUSTIC_EH)    // High E

// STRING TRANSITIONS AS COUNTER VALUES
#define TRANSITION_E_ETOP (OFFSET_E + (OFFSET_E - OFFSET_A)/2)
#define TRANSITION_E_A (OFFSET_E - (OFFSET_E - OFFSET_A)/2)
#define TRANSITION_A_D (OFFSET_A - (OFFSET_A - OFFSET_D)/2)
#define TRANSITION_D_G (OFFSET_D - (OFFSET_D - OFFSET_G)/2)
#define TRANSITION_G_B (OFFSET_G - (OFFSET_G - OFFSET_B)/2)
#define TRANSITION_B_EH (OFFSET_B - (OFFSET_B - OFFSET_EH)/2)
#define TRANSITION_EH_BOTTOM (OFFSET_EH - (OFFSET_B - OFFSET_EH)/2 )

// SMALLEST TIMER INCREMENT FOR A LED WITHIN A TRANSITION
#define LED_OFFSET_E  ((TRANSITION_E_ETOP - TRANSITION_E_A) / 6)
#define LED_OFFSET_A  ((TRANSITION_E_A - TRANSITION_A_D) / 6)
#define LED_OFFSET_D  ((TRANSITION_A_D - TRANSITION_D_G) / 6)
#define LED_OFFSET_G  ((TRANSITION_D_G - TRANSITION_G_B) / 6)
#define LED_OFFSET_B  ((TRANSITION_G_B - TRANSITION_B_EH) / 6)
#define LED_OFFSET_EH  ((TRANSITION_B_EH - TRANSITION_EH_BOTTOM) / 6)

volatile uint8_t signal;
volatile uint16_t timer;

uint16_t topCount;    // HOW MANY PERIODS
uint16_t timer_prev;  // OLD TCNT1
uint16_t result;      // SMALLEST SIGNAL PERIOD


// SIGNAL STABILIZING VARIABLES
unsigned int freqCount;
unsigned int burnCount;
short int burned;

unsigned short burnResultCount;
short int burned_result_mode;

// STRING TARGET INFORMATION VARIABLES
short int current_string;
```

```
short int current_string_offset;
short int prev_string_offset;
short int current_string_target;

void initCounter1() {

  // Counter register is 8-bit, and 256 is max value
  // so from above 0,032 ms * 256 increments = 8,192 ms
  // at 8,192 ms timer will overflow

  // TIMER/COUNTER1 PRESCALER
  // TCCR1B|=(1<<CS12)|(1<<CS10);  // 1024
     TCCR1B|=(1<<CS12);            // 256
  // TCCR1B|= (1<<CS11)|(1<<CS10); // 64

   // Enable Overflow Interrupt Enable
   TIMSK|=(1<<TOIE0);

   // INITIALIZE VARIABLES
   topCount = 0;
   burnCount = 0;
   freqCount = 0;
   burned_result_mode = 0;

   signal = 0;
   result = 0;
   burned = 0;
   current_string = 0;
   current_string_offset = 0;
   current_string_target = 0;
   prev_string_offset = 0;
}

void initCounter0() {

  // Counter register is 8-bit, and 256 is max value
  // so from above 0,032 ms * 256 increments = 8,192 ms
  // at 8,192 ms timer will overflow

  // TIMER/COUNTER0 PRESCALER
  // TCCR0|=(1<<CS02)|(1<<CS01)|(1<<CS00); // 1024
     TCCR0|=(1<<CS02);                     // 256
  // TCCR0|= (1<<CS01)|(1<<CS00);          // 64

  //Enable Overflow Interrupt Enable
  TIMSK|=(1<<TOIE0);
}

void initLEDS() {
    // PORTD SHALL SEND OUTPUT TO LEDS
    DDRD = 0xff;
}
```

```
void initADC() {

  /*
  ADC PRESCALER CALCULATION TABLE

  Prescaler / MHz
  256 / 8 000 000 = 0,000032 s = 0,032 ms

  MHz / Prescaler
  8 000 000 / 256 = 31 250 Hz = 31,250 kHz
  8 000 000 / 128 = 62 500 Hz = 62,500 kHz
  8 000 000 / 64 = 125 000 Hz = 125,000 kHz
  */

  // ADC PRESCALER
     ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0); // 128
  // ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (0 << ADPS0); // 64
  // ADCSRA |= (1 << ADPS2) | (0 << ADPS1) | (1 << ADPS0); // 32
  // ADCSRA |= (1 << ADPS2) | (0 << ADPS1) | (0 << ADPS0); // 16

  ADMUX |= (1 << REFS0); // Set ADC reference to AVCC
  ADMUX |= (1 << ADLAR); // Left adjust ADC result to allow easy 8 bit
      reading

  // ADC0 USED AS STANDARD, MUX SETTING IS SKIPPED

  ADCSRA |= (1 << ADATE);  // Set ADC to Free-Running Mode

  // ADC NOISE CANCELER - OPTIONAL
  // MCUCR |= (1 << SM0); // Setting sleep mode to "ADC Noise Reduction
      "
  // MCUCR |= (1 << SE);  // Sleep enable

  ADCSRA |= (1 << ADEN);  // Enable ADC
  ADCSRA |= (1 << ADIE);  // Enable ADC Interrupt
}

int main (void) {

   initLEDS();
   initADC();
   initCounter1();
   initCounter0();

   sei();   // Enable Global Interrupts

   ADCSRA |= (1 << ADSC);  // Start A2D Conversions

   while(1) { // Loop Forever

    // ADC NOISE CANCELER - EXPERIMENTAL
    /*
    MCUCR |= (1<<SE);         // Set enable sleep
    ADCSRA |= (1<<ADSC);      // Start converting
```

```
    __asm volatile ("sleep"); // go to sleep
    MCUCR &= ~(1<<SE);        // Clear enable sleep
    */
   }
}

ISR(TIMER1_OVF_vect) {

  // This is the interrupt service routine for TIMER0 OVERFLOW
      Interrupt.
  // CPU automatically call this when TIMER0 overflows.

  // WHEN TIMER1-COUNTER REACHES OVERFLOW ALL ALGORITHM VARIABLES WILL
      BE RESET
  // RESET BURN
  burnCount = 0;
  burnResultCount = 0;
  burned = 0;

  // RESET TOP COUNT
  topCount = 0;
  result = UINT16_MAX;
  timer_prev = 0;
  timer = 0;
  TCNT1 = 0;
}

ISR(ADC_vect) {

  // SAVE ADC' VALUE, AND THE TIMER1 VALUE
  signal = ADCH;
  timer = TCNT1;

  // IF SIGNAL IS LOWER THAN result = 0xFFFF -> SIGNAL IS SET TO RESULT
  // RESULT WILL BE DECREMENTED PROPORTIONALLY TO SIGNAL
  if(signal < result){
    result = signal;
  }

  // IF NO TOPS AND WE ARE AT BEGINNING OF COUNT
  if(topCount == 0) {
    TCNT1 = 0;
    timer = 0;
  }

  // WHEN SIGNAL RISES AGAIN, IS LARGER THAN result, THEN A PERIOD IS
      REGISTERED
  // CONDITION HAS A NOISE THRESHOLD
  if(signal > result && signal < SIGNAL_CUTOFF_THRESHOLD) {

    // ONCE SIGNAL IS LARGER THAN SIGNAL_CUTOFF_THRESHOLD, THEN
        INITIATE BURN SEQUENCE
    // BURN NOTIFIES HOW MANY PERIODS SHALL BE SKIPPED
    if(burnCount < BURN && burned == 0) {
```

```
    ++burnCount;
    return;
  } else {

    // WHEN BURN SEQUENCE IS FINISHED SIGNAL IS "STABLE"
    // PERIOD COUNTER IS INITIATED, BURN SEQUENCE VARIABLES ARE RESET
    if(burnCount >= BURN && burned == 0) {
      burnCount = 0;
      burned = 1;
      TCNT1 = 0;
    }

    // IF TIME INTERVAL BETWEEN PERIODS IS TOO BIG, THEN RESET
        ALGORITHM
    if((timer - timer_prev) > 2000) {
      topCount = 0; // SUBSEQUENTLY RESETS TCNT1 AND timer
      result = UINT16_MAX;
      TCNT1 = 0;    // JUST IN CASE
      timer_prev = 0;
      return;
    }

    // A TOP IS FOUND
    topCount++;

  }

  // SAVE TCNT1 FOR DETECTION OF ERRONEOUS TIME INTERVALS BETWEEN
      PERIODS
  timer_prev = timer;

  // RESET RESULT
  result = UINT16_MAX;
}

// IF BURN SEQUENCE, AND, PERIOD COUNT IS FINISHED, THEN FREQUENCY IS
     FOUND
if(topCount == SAMPLES) {

    // TIMER1 REPRESENTS THE FREQUENCY, SEE DECLARATIONS
    result = TCNT1;

    // E
    if( TRANSITION_E_ETOP > result && result > TRANSITION_E_A) {
      current_string = 0;

      if(result >= (TRANSITION_E_A)) {
        current_string_offset = 6;
      }
      if(result >= (TRANSITION_E_A + LED_OFFSET_E)) {
        current_string_offset = 5;
      }
      if(result >= (TRANSITION_E_A + (LED_OFFSET_E*2))) {
        current_string_offset = 4;
```

```
  }
  if(result >= (TRANSITION_E_A + (LED_OFFSET_E*3))) {
    current_string_offset = 3;
  }
  if(result >= (TRANSITION_E_A + (LED_OFFSET_E*4))) {
    current_string_offset = 2;
  }
  if(result >= (TRANSITION_E_A + (LED_OFFSET_E*5))) {
    current_string_offset = 1;
  }
  if(result >= (TRANSITION_E_A + (LED_OFFSET_E*6))) {
    current_string_offset = 0;
  }
}

// A
if( TRANSITION_E_A > result && result > TRANSITION_A_D) {
  current_string = 1;

  if(result >= (TRANSITION_A_D)) {
    current_string_offset = 6;
  }
  if(result >= (TRANSITION_A_D + LED_OFFSET_A)) {
    current_string_offset = 5;
  }
  if(result >= (TRANSITION_A_D + (LED_OFFSET_A*2))) {
    current_string_offset = 4;
  }
  if(result >= (TRANSITION_A_D + (LED_OFFSET_A*3))) {
    current_string_offset = 3;
  }
  if(result >= (TRANSITION_A_D + (LED_OFFSET_A*4))) {
    current_string_offset = 2;
  }
  if(result >= (TRANSITION_A_D + (LED_OFFSET_A*5))) {
    current_string_offset = 1;
  }
  if(result >= (TRANSITION_A_D + (LED_OFFSET_A*6))) {
    current_string_offset = 0;
  }
}

// D
if( TRANSITION_A_D > result && result > TRANSITION_D_G) {
  current_string = 2;

  if(result >= (TRANSITION_D_G)) {
    current_string_offset = 6;
  }
  if(result >= (TRANSITION_D_G + LED_OFFSET_D)) {
    current_string_offset = 5;
  }
  if(result >= (TRANSITION_D_G + (LED_OFFSET_D*2))) {
    current_string_offset = 4;
```

```
  }
  if(result >= (TRANSITION_D_G + (LED_OFFSET_D*3))) {
    current_string_offset = 3;
  }
  if(result >= (TRANSITION_D_G + (LED_OFFSET_D*4))) {
    current_string_offset = 2;
  }
  if(result >= (TRANSITION_D_G + (LED_OFFSET_D*5))) {
    current_string_offset = 1;
  }
  if(result >= (TRANSITION_D_G + (LED_OFFSET_D*6))) {
    current_string_offset = 0;
  }
}

// G
if( TRANSITION_D_G > result && result > TRANSITION_G_B) {
  current_string = 3;

  if(result >= (TRANSITION_G_B)) {
    current_string_offset = 6;
  }
  if(result >= (TRANSITION_G_B + LED_OFFSET_G)) {
    current_string_offset = 5;
  }
  if(result >= (TRANSITION_G_B + (LED_OFFSET_G*2))) {
    current_string_offset = 4;
  }
  if(result >= (TRANSITION_G_B + (LED_OFFSET_G*3))) {
    current_string_offset = 3;
  }
  if(result >= (TRANSITION_G_B + (LED_OFFSET_G*4))) {
    current_string_offset = 2;
  }
  if(result >= (TRANSITION_G_B + (LED_OFFSET_G*5))) {
    current_string_offset = 1;
  }
  if(result >= (TRANSITION_G_B + (LED_OFFSET_G*6))) {
    current_string_offset = 0;
  }
}

// B
if( TRANSITION_G_B > result && result > TRANSITION_B_EH) {
  current_string = 4;

  if(result >= (TRANSITION_B_EH)) {
    current_string_offset = 6;
  }
  if(result >= (TRANSITION_B_EH + LED_OFFSET_B)) {
    current_string_offset = 5;
  }
  if(result >= (TRANSITION_B_EH + (LED_OFFSET_B*2))) {
    current_string_offset = 4;
```

```
  }
  if(result >= (TRANSITION_B_EH + (LED_OFFSET_B*3))) {
    current_string_offset = 3;
  }
  if(result >= (TRANSITION_B_EH + (LED_OFFSET_B*4))) {
    current_string_offset = 2;
  }
  if(result >= (TRANSITION_B_EH + (LED_OFFSET_B*5))) {
    current_string_offset = 1;
  }
  if(result >= (TRANSITION_B_EH + (LED_OFFSET_B*6))) {
    current_string_offset = 0;
  }
}

// EH
if( TRANSITION_B_EH > result && result > TRANSITION_EH_BOTTOM) {
  current_string = 5;

  if(result >= (TRANSITION_EH_BOTTOM)) {
    current_string_offset = 6;
  }
  if(result >= (TRANSITION_EH_BOTTOM + LED_OFFSET_EH)) {
    current_string_offset = 5;
  }
  if(result >= (TRANSITION_EH_BOTTOM + (LED_OFFSET_EH*2))) {
    current_string_offset = 4;
  }
  if(result >= (TRANSITION_EH_BOTTOM + (LED_OFFSET_EH*3))) {
    current_string_offset = 3;
  }
  if(result >= (TRANSITION_EH_BOTTOM + (LED_OFFSET_EH*4))) {
    current_string_offset = 2;
  }
  if(result >= (TRANSITION_EH_BOTTOM + (LED_OFFSET_EH*5))) {
    current_string_offset = 1;
  }
  if(result >= (TRANSITION_EH_BOTTOM + (LED_OFFSET_EH*6))) {
    current_string_offset = 0;
  }
}

// DETECT TARGET STRING, SAVE IT, AND ENTER BURN RESULT MODE
if(burned_result_mode == 0) {

  // INITIATE BURN RESULT VARIABLES
  burned_result_mode = 1;
  burnResultCount = 0;

  // STRING TARGET CHANGE
  current_string_target = current_string;

  // INDICATE STRING CHANGE ON LEDS (NEW STRING TARGET)
  for(int i = 0; i < 7; i++) {
```

```
    PORTD ^= (1 << current_string_target);
    _delay_ms(150);
  }

  // SHOW FIRST VALUE AS LED INDICATION WILL OTHERWISE BE SKIPPED
  PORTD = 0;
  PORTD |= (1 << current_string_offset);

} else if(burned_result_mode == 1) {

  // IN BURN RESULT MODE DO THE FOLLOWING
  // BURN NOT COMPLETE, AND, STRING IS NOT TARGET

  // CHECK IF WE ARE ON TARGET, IF SO, INDICATE STRING WITH LEDS
  if(current_string == current_string_target) {

    // OUTPUT TO LEDS, CORRECT STRING
    if(current_string_offset < prev_string_offset) {
      for(int i = prev_string_offset ; i > (current_string_offset
           - 1); --i) {
        PORTD = (1 << i);
        _delay_ms(70);
      }
    } else if(current_string_offset > prev_string_offset) {
      for(int i = prev_string_offset; i < (current_string_offset
          + 1); ++i) {
        PORTD = (1 << i);
        _delay_ms(70);
      }
    }
    PORTD = 0;
    PORTD |= (1 << current_string_offset);

    prev_string_offset = current_string_offset;

    // RESET OUR BURN COUNTER TO PROCEED WITH THE BURN ALGORITHM
    burnResultCount = 0;
  } else {
    // IF OFF TARGET, INCREMENT BURN COUNT
    burnResultCount++;
    PORTD = 0;
  }

  // CHECK IF BURN SEQUENCE HAS COMPLETED
  if(burnResultCount == BURN_RESULT){

    // RESET VARIABLES
    burned_result_mode = 0;
  }
}

// RESET VARIABLES FOR NEXT PERIOD COUNT
result = UINT16_MAX;
topCount = 0;
```

```
      TCNT1 = 0;

      // RESET BURN SEQUENCE
      burned = 0;
    }
}
```