Design av et enkelt FSK-kommunikasjonssystem

Forfatter: Fredrik Ellertsen
Versjon: 3
Dato: 25.11.2015
Kontrollert av:
Dato:

Innhold

1 Innledning 1

2 Mulig løsning 1

3 Realisering og test 3
3.1 Programkodens virkemåte 4
3.2 Uventede problemer som oppstod 6

4 Konklusjon 6

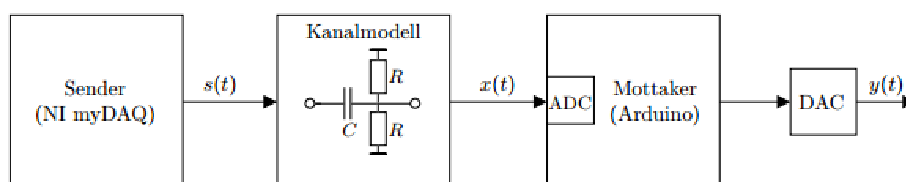
5 Takk 6

6 Vedlegg A: kildekode 7

TTT4265 – ESDA II – NTNU 2015

1 Innledning

Digital kommunikasjon foregår vanligvis binært. Utover dette er det store forskjeller på hvilke protokoller som anvendes i ulike kommunikasjonsplattformer. Dette designnotatet vil ta for seg såkalt *frequency-shift keying* (FSK), som brukes i bl.a. Bluetooth-kommunikasjon. FSK bruker samme bølgeform for nullere og enere, men med ulike frekvenser. I dette notatet vil et mulig design av en FSK-mottaker presenteres, samt at en FSK-sender vil spesifiseres. Et slikt system er illustrert på blokkskjemanivå i Figur 1.



Figur 1: Blokkskjema av et enkelt FSK-system med sender, kanalmodell og mottaker [2].

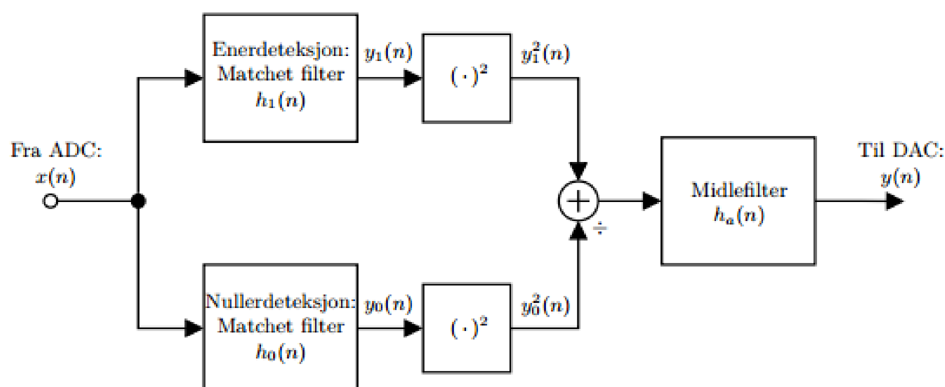
De informasjonsbærende enhetene vil i dette designet være firkantpulstog der logisk 0 vil representeres ved en frekvens f_0 og logisk 1 vil representeres ved en annen frekvens f_1. Kravsspesifikasjonen nevner at symbollengden T_{sym} skal være $0,111$ sekunder, og at samplingsraten f_s skal være omtrent $^{60}/_{T_{sym}} \approx 540,5$ Hz. Symbollengden angir hvor lenge senderen skal holde frekvensen som representerer ett symbol før den går videre til neste.

For å gjøre designproblemet mer realistisk antas det at kanalen vi bruker har høypassegenskaper (f.eks. kroppens kapasitans), og kanalmodellen i Figur 1 er derfor beskrevet med et høypassfilter med krav om knekkfrekvens $f_c = 49$ Hz. Hovedbolken av arbeidet i dette designet går ut på å spesifisere og implementere programkoden som skal kjøre på mottakeren (Arduino, [1]).

Systemet vil testes ved å se på responsen når systemet sender/mottar ulike bitstrømmer og avgjøre hvorvidt det er mulig å tyde utgangssignalet med rimelig nøyaktighet.

2 Mulig løsning

En mulig måte å implementere deteksjonsalgoritmen på Arduinoen baserer seg på blokkskjemaet i Figur 2.



Figur 2: Blokkskjema av en mulig måte å strukturere deteksjonsalgoritmen [2].

Dette blokkskjemaet illustrerer *asynkron* FSK, som betyr at det ikke finnes noen klokke som synkroniserer senderen og mottakeren. Det er mulig å konstruere et synkront FSK-system, men da er det behov for å inkludere en form for synkroniseringsdata før selve dataene overføres. Slik kommunikasjon er utenfor skop av dette designnotatet, og designet har derfor grunnlag i Figur 2.

Systemet i Figur 2 utnytter det faktum at vi kjenner formen på gyldige signal, og at vi da kan sample inngangssignalet $x(t)$ med $f_s = 540,5$ Hz og konvolvere de 60 forrige sample-verdiene med to ulike matchede filtre h_1 og h_0. Filtrene kan konstrueres ved å studere sekvensen av samplede verdier og dermed lage to lister med 60 koeffisienter med samme form som de gyldige signalene. Når systemet mottar et gyldig symbol vil ett av filtrene gi stort utslag, og ved å kvadrere utgangen av hvert filter, finne differansen mellom dem og midle resultatet over en hel samplingsperiode sitter vi i teorien igjen med et utgangssisgnal $y(n)$ som er nokså høyt for logisk 1, og nokså lavt for logisk 0.

Arduino-plattformen som skal benyttes vil sample ved hjelp av tidsstyrte avbrudd med samplingsintervall $T_s = {}^{0,111\text{ s}}/_{60} = 1850$ µs. Med en klokkefrekvens på 16 MHz betyr dette at løkken som skal sample og behandle verdiene må være forholdsvis rask.

Senderen vil realiseres ved hjelp av hardware-plattformen NI myDAQ og et tilpasset virtuelt instrument [3]. Slik kan sender-blokken i Figur 1 oversette en vilkårlig bitstrøm til et FSK-signal og sende dette signalet gjennom myDAQ-en til mottakeren.

Kanalmodellen kan realiseres ved hjelp av to motstander på $R = 8,2$ kΩ og en kondensator på $C = 394$ nF slik som i Figur 2. Knekkfrekvensen blir

da

$$f_c = \frac{1}{2\pi(R/2)C} \approx 49 \text{ Hz}$$

Kanalmodellen har også som funksjon å biase inngangssignalet slik at det alltid er positivt. Dette er nødvendig da Arduinoens ADC ikke kan arbeide med negative spenninger.

Frekvensene f_0 og f_1 er valgfrie, men bør oppfylle en håndfull krav. De må være hele multipler av $1/T_{sym}$ slik at senderen alltid sender et helt antall perioder av firkantsignalene i hver symbolperiode. Vi kan da stille følgende krav

$$\frac{f_s}{f_0} = \frac{60/T_{sym}}{k/T_{sym}} = \frac{60}{k} = 2m \in \mathbb{N} \tag{1}$$

og tilsvarende for f_1 der l står i nevneren. (1) sier med andre ord at halve signalperioden skal være en multippel av samplingsperioden T_s. Filtrene blir da symmetriske og enkle å implementere. Ved hjelp av Nyquists samplingsteorem samt den oppgitte knekkfrekvensen kan vi låse frekvensene slik:

$$f_0, f_1 \in [49 \text{ Hz}, 270 \text{ Hz}] \tag{2}$$

Jo lavere frekvensene er, jo flere samples får vi plass til på én symbolperiode, og dermed blir mottakeren mer robust. Ved å velge $k = 10$ og $l = 6$ får vi

$$f_0 = 6/60 \cdot 540,5 \text{ Hz} = 54,05 \text{ Hz}$$

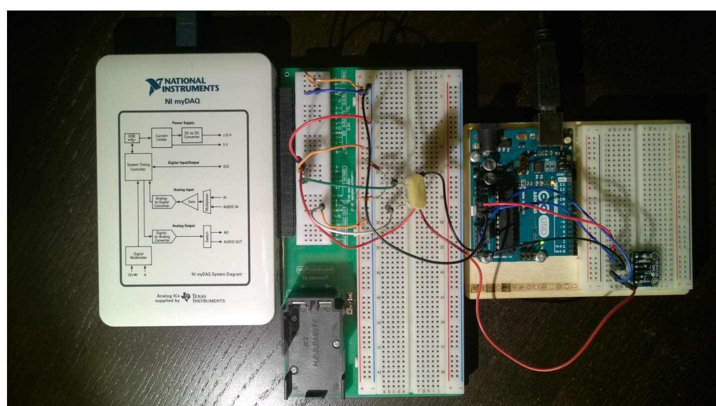
$$f_1 = 10/60 \cdot 540,5 \text{ Hz} \approx 90,09 \text{ Hz}$$

Frekvensen f_0 ligger nokså nærme f_c og vil nok dempes en del, men kan enkelt skaleres opp igjen i differanse-blokken i algoritmen.

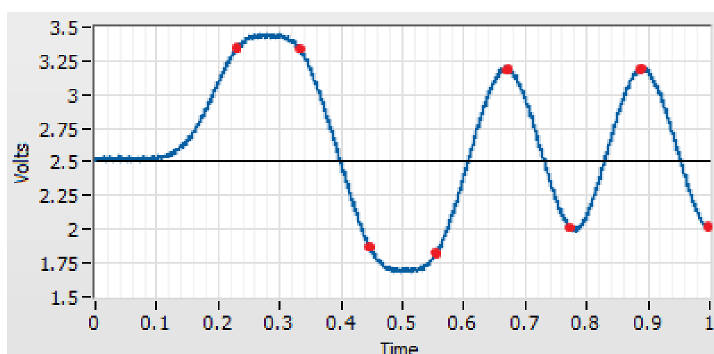
3 Realisering og test

Kretsen ble koblet opp og er avbildet i Figur 3. Arduino-en ble programmert og kildekode finnes i Vedlegg A. DAC-modulen som ble benyttet er MCP4725 [4].

Figur 4 viser utgangssignalet $y(t)$ når senderen sender bitstrømmen "11001010". Som man ser av figuren er det enkelt å tyde utgangssignalet $y(n)$. Amplituden til $y(n)$ varierer mye fra gang til gang, noe som er avhengig av når samplingen av signalet finner sted. I Figur 5 ligger en oscilloskopskjermdump som viser en mulig fase mellom inngangssignalet $x(t)$ og samplede verdier $x(n)$. Andre faser vil gi andre verdier for $y(t)$, men formen vil være den samme. Dette problemet har rot i systemets asynkrone natur og er vanskelig å ordne opp i.



Figur 3: Oppkoblet krets.

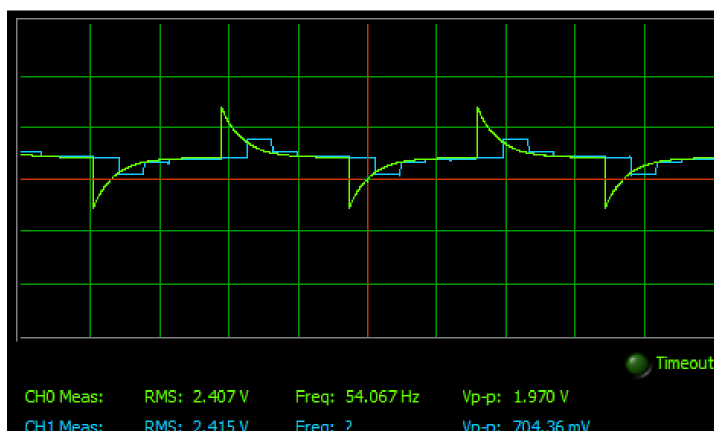


Figur 4: Skjermdump av utgangssignalet $y(t)$ når mottakeren mottar bitstrømmen "11001010". De gyldige verdiene av $y(t)$ ligger på 0,111 s, 0,222 s osv. og er markert med rødt i bildet. Man ser at det er mulig å dekode bitstrømmen ved kun å se på utgangssignalet. Signalet fra myDAQ-en har en amplitude på omtrent 1 V.

3.1 Programkodens virkemåte

Arduinoen har flere innebygde timere som kan konfigureres på ulike måter. Det finnes forhåndsdefinerte biblioteker for å sette opp tidsstyrte avbrudd, men i dette designet ble lavnivå registermanipulasjon benyttet for å ha full kontroll over avbruddsfrekvensen. Her brukes *timer1* i CTC-modus (compare match mode) slik at en avbruddsrutine kalles hver gang timer-registeret når en forhåndssatt verdi CM. Dette avbruddet setter et flagg som fanges i hovedløkken slik at sampling skjer til korrekt tid. Denne måten å sample på er nødvendig på Arduino siden DAC-en snakker I^2C og benytter derfor avbrudd for å sende data. Avbrudd inni avbrudd fungerer svært dårlig, så et flagg må brukes.

timer1 sitt compare-register rommer 16 bit (0 - 65535, til forskjell fra de to andre som rommer en byte hver). Vi kan med andre ord oppnå høyere



Figur 5: Skjermdump av inngangssignalet $x(t)$ (grønn) og samplede verdier $x(n)$ i én mulig fase. Slike oscilloskopmålinger var også grunnlaget for de matchede filtrene h_1 og h_0.

presisjon ved å bruke *timer1*. Arduinoens systemklokke kjører på 16 MHz, og uten å preskalere klokken får vi

$$CM = \frac{16 \text{ MHz}}{540,5 \text{ Hz}} \approx 29600 < 65535$$

som gir en avbruddsfrekvens på $540,5$ Hz. Programmering er som kjent en svært iterativ prosess, og avbruddsfrekvensen ble testet ved at programkoden sendte de samplede verdiene rett til DAC-en uten filtrering som i Figur 5. CM-verdien ble finpusset til utgangssignalet ikke lenger skled i forhold til inngangssignalet. $CM = 29582$ fungerte bra.

Arduinoen har en innebygget ADC med 10 bits oppløsning. Når loopen fanger flagget som settes av avbruddsrutinen sampler den én verdi til variabelen *val* og trekker fra 495 så signalet ligger omtrent rundt null. Deretter settes den samplede verdien inn det globale ringbufferet *sampleBuffer* gjennom funksjonen *pushBuffers()*. Deretter kalles funksjonen *applyCoeffs()* som konvolverer to matchede filtre *LFcoeffs* og *HFcoeffs* med listen av samples og legger resultatene hhv. i *y0* og *y1*. Resultatet nedskaleres til slutt for å unngå overflyt. I samme funksjon legges også differansen mellom de kvadrerte resultatene inn i et ringbuffer. Dette er gjort slik at det er mulig å både gjennomføre konvolusjon og midle utgangssignalet i én for-løkke for å spare tid. Midlingen av utgangssignalet gjøres ved hjelp av koeffisientlisten *triangleAverageFilter* som har trekantet impulsrespons. En slik koeffisientliste gir glattere midling enn et rent gjennomsnitt. Flere typer midlefiltre ble forsøkt, men trekantformen så ut til å gi best resultat.

Det faktum at samme for-løkke arbeider på både samples-listen og differanse-listen gjør at utgangssignalet er forsinket med 61 samples. Dette er ikke

et problem da kommunikasjonen allikevel er asynkron. Den midlede og nedskalerte summen av differansebufferet trekkes til slutt opp med 2048 trinn slik at det kan sendes til en 12 bits DAC. Differansen mellom de kvadrerte signalene er også skalert for å kompensere for høypassegenskapene til kanalen.

3.2 Uventede problemer som oppstod

Det var store problemer i tidligere versjoner av programkoden med at kjøretiden var for lang. Dette ble løst ved å gjøre all filtrering i samme for-løkke samt ved å øke DAC-ens samplingsfrekvens. I tillegg gjøres all skalering utenfor for-løkken. Slik ble koden til slutt rask nok til at kjøretiden lå under samplingsintervallet. Koden i hovedløkken bruker nå omtrent 1400 µs på å kjøre én gang.

Det var også nokså krevende å finne fornuftige skaleringskoeffisienter. Røffe overslag ga en pekepinn på hvor stor skaleringsfaktor som måtte til i de ulike delene, men de fleste koeffisientene ble funnet ved å prøve og feile. Det ble funnet at ved å skalere med potenser av to unngår Arduinoens prosessor flyttallsdivisjon, noe som reduserer kjøretiden betraktelig.

4 Konklusjon

Dette designnotatet har beskrevet design av en FSK-mottaker samt spesifikasjon av en FSK-sender. Systemet benytter seg av firkantpulser med frekvensene $f_0 = 54,05$ Hz og $f_1 = 90,09$ Hz for å kode hhv. nullere og enere. Systemet består i hovedsak av et digitalt filter som benytter seg av tre ulike FIR-filtre for å detektere de ulike frekvensene. Mottakeren i kretsen gir ut et signal som man kan dekode uten store problemer, og oppfyller derfor alle kravene som ble stilt.

5 Takk

Jeg vil gjerne takke vitenskapelig personell ved Institutt for elektronikk og telekommunikasjon, NTNU for frembringelse av Figur 1 og 2 samt for utvikling av det tilpassede virtuelle instrumentet fsksender.exe.

Referanser

[1] Arduino-plattformen:
www.arduino.cc/

[2] T. Ekman: Designøving 4 i emne TTT 4265 Elektronisk systemdesign og -analyse II, NTNU, 2015

[3] National Instruments' myDAQ med tilhørende programvare labVIEW:
http://www.ni.com/mydaq/

[4] MCP4725 DAC med 12 bit oppløsning og I^{2}C-interface:
https://www.adafruit.com/products/935

6 Vedlegg A: kildekode

Listing 1: Kildekode

```
1  #include <Wire.h>
2  #include <Adafruit_MCP4725.h>
3
4  Adafruit_MCP4725 dac; // initialize DAC object
5
6  const int bufferLength = 60; // length of buffers
7
8  // coeffs for detecting HF signal (90.09 Hz) (absolute sum = 156)
9  const int8_t LFcoeffs[bufferLength] = {
10     8,3,1,1,0,-8,-3,-1,-1,0,
11     8,3,1,1,0,-8,-3,-1,-1,0,
12     8,3,1,1,0,-8,-3,-1,-1,0,
13     8,3,1,1,0,-8,-3,-1,-1,0,
14     8,3,1,1,0,-8,-3,-1,-1,0,
15     8,3,1,1,0,-8,-3,-1,-1,0
16 };
17
18 // coeffs for detecting LF signal (54.054 Hz) (absolute sum = 160)
19 const int8_t HFcoeffs[bufferLength] = {
20     6,2,0,-6,-2,0,
21     6,2,0,-6,-2,0,
22     6,2,0,-6,-2,0,
23     6,2,0,-6,-2,0,
24     6,2,0,-6,-2,0,
25     6,2,0,-6,-2,0,
26     6,2,0,-6,-2,0,
27     6,2,0,-6,-2,0,
28     6,2,0,-6,-2,0,
29     6,2,0,-6,-2,0
30 };
31
32 // triangular average filter
```

```
33 const uint8_t triangleAverageFilter[bufferLength]{
34     8, 16, 24, 32, 40, 48,
35     56, 64, 72, 80, 88, 96,
36     104, 112, 120, 128, 136, 144,
37     152, 160, 168, 176, 184, 192,
38     200, 208, 216, 224, 232, 240,
39     232, 224, 216, 208, 200, 192,
40     184, 176, 168, 160, 152, 144,
41     136, 128, 120, 112, 104, 96,
42     88, 80, 72, 64, 56, 48,
43     40, 32, 24, 16, 8, 0
44 };
45
46 long output = 0; // output value
47 long diff = 0; // temporary value
48 int xn; // stores sampled values
49
50 int bufferPointer = 0; // start pointer in ring buffers
51 int sampleBuffer[bufferLength]; // contains samples
52 long diffBuffer[bufferLength]; // contains squared difference
53
54 // stores filtered values
55 long y0 = 0;
56 long y1 = 0;
57
58 volatile bool trigger = false; // interrupt flag for the loop
59
60 void setup() {
61     // timer compare match setup (540,54 Hz)
62     TCCR1A = 0; // clears registers
63     TCCR1B = 0;
64     OCR1A = 29582; // this value gives no drift
65     TCCR1B |= (1 << WGM12); // CTC mode on
66     TCCR1B |= (0 << CS12) | (0 << CS11) | (1 << CS10); // prescaler: none
67     TIMSK1 |= (1 << OCIE1A);  // enable timer compare interrupt
68
69     // increase DAC sample frequency
70     ADCSRA |= (1 << ADPS1);
71     ADCSRA &= ~((1 << ADPS2) | (1 << ADPS0));
72
73     // initialize I2C communication with DAC object
74     dac.begin(0x62);
75 }
76
77 void loop() {
78     if (trigger) {
79         trigger = false; // reset trigger flag
80         xn = analogRead(A0) - 495; // pulls down the sampled value to 0,
81         pushBuffers(); // updates buffers with xn and diff
82         applyCoeffs(); // updates y0, y1 and diffBuffer
83         dac.setVoltage(output + 2048, false); // shifts and writes to DAC
84     }
85 }
86
```

```
87  long outputmax = 0;
88  void applyCoeffs() {
89      // square the filtered values and find the difference,
90      // scaling to compensate for the HPF channel filter
91      diff = (y0 * y0) * 5 - (y1 * y1) * 3;
92
93      // resets filtered variables
94      y0 = 0;
95      y1 = 0;
96
97      for (uint8_t i = 0; i < bufferLength; i++) {
98          uint8_t pos = (bufferPointer + i) % bufferLength;
99
100         // sum differences and weight the average
101         output += diffBuffer[pos] * triangleAverageFilter[i];
102
103         // weights filtered values
104         y0 += (LFcoeffs[i] * sampleBuffer[pos]);
105         y1 += (HFcoeffs[i] * sampleBuffer[pos]);
106     }
107
108     // scales down filtered values
109     y0 /= 512;
110     y1 /= 512;
111
112     output /= 16384; // averages and downscales output
113 }
114
115 void pushBuffers() {
116     bufferPointer--;
117
118     if (bufferPointer < 0) {
119         bufferPointer = bufferLength - 1; // loop back if underflow
120     }
121
122     sampleBuffer[bufferPointer] = xn; // updates with current sample
123     diffBuffer[bufferPointer] = diff; // updates with previous difference
124 }
125
126 ISR(TIMER1_COMPA_vect) {
127     // set trigger flag for the loop to catch
128
129     trigger = true;
130 }
```