NTNU

elsys
ELEKTRONISK SYSTEMDESIGN & INNOVASJON

Designnotat

Tittel: Design av FSK-demodulator	
Forfattere: Hans Jakob Vahlin	
Versjon: 3.0	Dato: 02.12.18

Innhold

---

1 Innledning

Digital kommunikasjon kan brytes ned til en sender som sender en bitsekvens til en mottaker. En enkel måte og oppnå dette på er og koble avsender og mottaker sammen med et ideelt lederpar. Ved og la spenningen til avsender variere mellom logisk `HØY` og logisk `LAV` kan en bitsekvens reproduseres hos mottaker. I tilfeller hvor et ideelt lederpar ikke er tilgjengelig, f.eks i radiokommunikasjon, kan en bitsekvens reproduseres ved FSK (*Frequency Shift Keying*) modulasjon. Her representeres de logiske nivående ved at avsender skifter mellom og sende et signal med høy frekvens og lav frekvens. [1]

I dette designnotatet skal det diskuteres implementasjonen av en FSK-demodulator. Det overordnete systemet er vist i Figure 1.

Figur 1: Overordnet system for en FSK-demodualtor.

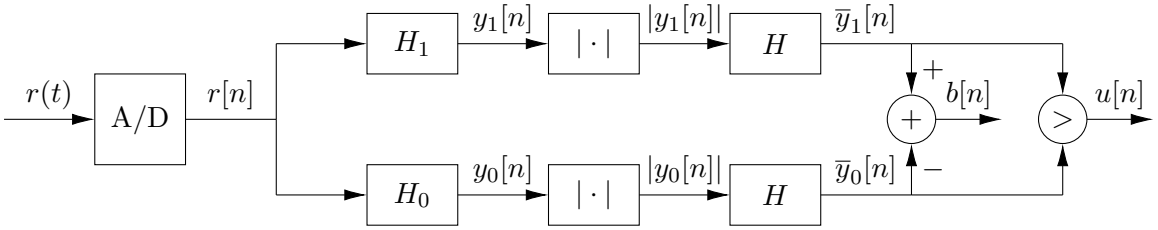
Inngangssignalet $r(t)$ er et sinusformet signal som enten har frekvensen f_0 eller f_1, hvor $f_0 < f_1$. Det ene utgangssignalet, $b(t)$, skal monitorere om det er f_0 eller f_1 som kommer inn. $b(t)$ skal være logisk `HØY` når $r(t)$ består av frekvenskomponenten f_0, og logisk `LAV` når $r(t)$ består av frekvenskomponenten f_1. Det andre utgangssignalet $u(t)$, skal monitorere hvorvidt $r(t)$ er tilstede på inngangen. Dersom $r(t)$ er tilstede, skal $u(t)$ være logisk `HØY`, og dersom $r(t)$ ikke er tilstede, skal $u(t)$ være logisk `LAV`.

Demodulatoren som blir implementert skal ha et areal på mindre enn 4cm^2

2 Prinsipiell løsning

2.1 Oversikt

En muling implementasjon av FSK-demodulatoren beskrevet i kapittel 1 er vist i Figure 2.



Figur 2: Blokkskjema for FSK-demodulator.

Inngangssignalet $r(t)$ blir punktprøvd med en samplingsfrekvens $f_s > 2 \cdot f_1 > 2 \cdot f_0$, slik at Nyqvists teorem blir oppfylt. (Konverteringen fra analog til digital vil ikke bli diskutert i dette designnotatet, men det antas at A/D omformeren har god nok oppløsning til at det kan sees bort ifra kvantifiseringsfeil.) Det digitale signalet, $r[n]$, blir da $r[n] = r(nT_s)$, hvor T_s er perioden til samplingsfrekvensen. Videre blir det digitale signalet sendt inn til to filter. Et høypassfilter, H_1 og et lavpassfilter, H_0. Hensikten her er at dersom $r[n]$ består av en høy frekvens - dvs f_1, vil signalet som går gjennom høypassfilteret, $y_1[n]$ ha en større amplitude enn signalet som går gjennom lavpassfilteret, $y_0[n]$. Tilsvarende resonnement sier at amplitudene til $y_0[n]$ og $y_1[n]$ vil bli hhv. stor og liten ved lav frekvens - dvs f_0.

Neste modul returnerer absoluttverdien, $|y_i[n]| \ \forall \ i \in \{1, 2\}$, slik at en middelverdi, $\overline{y}_i \ \forall \ i \in \{1, 2\}$, kan beregnes basert på N samplete verdier. Verdien til N må være stor nok til at en fornuftig middelverdi kan beregnes, men samtidig ikke så stor at den gjør middelverdi filteret "tregt". Middelverdiene blir deretter sammenliknet. Dersom $\overline{y}_1 > \overline{y}_0$ betyr det at $r(t)$ må ha frekvenskomponent f_1, og $b[n]$ settes følgelig til logisk `HØY`. I tilfellene hvor $\overline{y}_0 > \overline{y}_1$ må frekvensen til $r(t)$ være f_0, og følgelig blir $b[n]$ satt til logisk `LAV`.

I tilegg blir middelverdiene sammenliknet med en terskelverdi. Dersom en av middelverdiene $\overline{y}_1[n]$ eller $\overline{y}_0[n]$ er større en terskelverdien, blir $u(t)$ logisk `HØY`. Hvis ikke blir $u(t)$ satt til logisk `LAV`. Terskelverdien det sammenliknes med vil være basert på målinger av hva $\overline{y}_1[n]$ og $\overline{y}_0[n]$ blir når signalet som kommer inn inneholder f_0 eller f_1. Terskelverdien vil bli nedjustert

litt fra målingsverdiene for og gi litt slingringsrom. Dersom en av middelverdiene er større enn denne verdien må signalet $r(t)$ være tilstede på inngangen.

2.2 Høypass og lavpass -filter

Til implementeringen av høypassfilteretet, H_1 og lavpassfilteret H_0 brukes et *Infinite Impulse Response* filter - IIR. Et IIR filter er et rekursivt filter hvor den utgående verdien, $y[n]$, vil være avhengig av den innkommende verdien, $r[n]$ og den forrige utgående verdien, gjengitt i likning (1).

$$y[n] = r[n] + a \cdot y[n-1] \ \ \forall \ \ a \in \{-1, 1\} \tag{1}$$

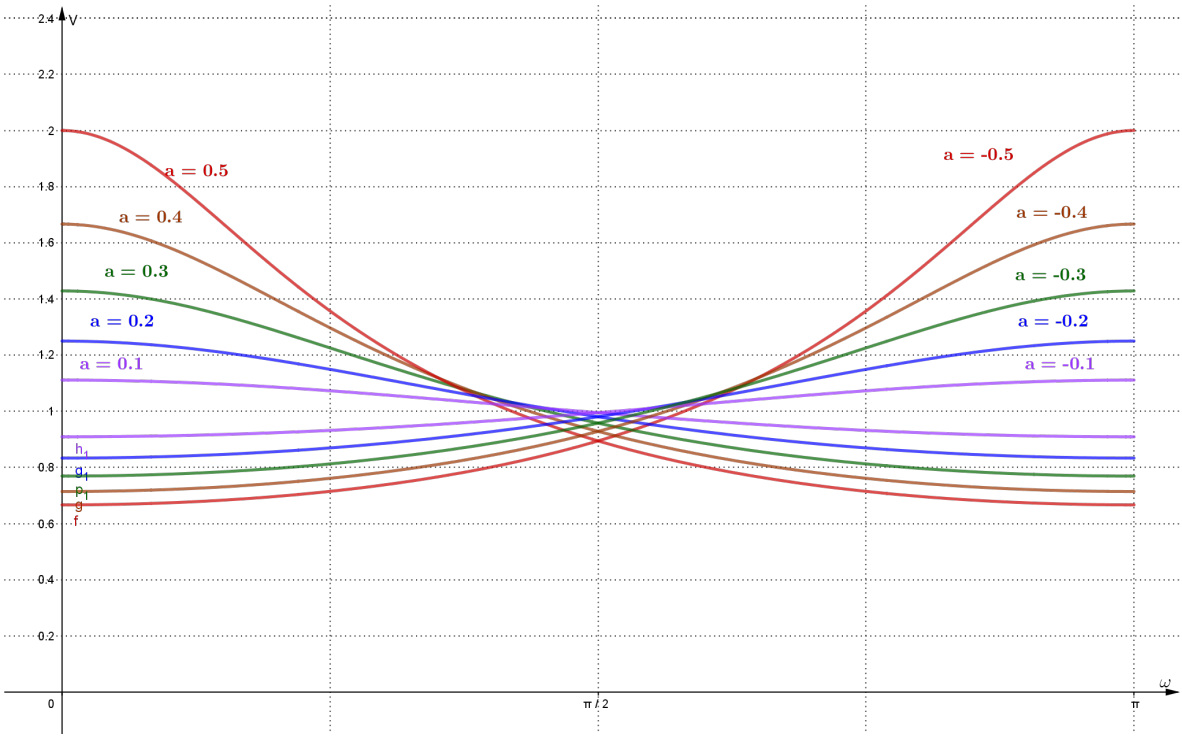
Det kan utledes at et slikt filter vil ha frekvensresponsen

$$H(j\omega) = \frac{1}{1 - ae^{-j\omega}} \ \ \forall \ \ a \in \{-1, 1\} \tag{2}$$

Og følgelig med en amplituderespons gitt ved

$$|H(j\omega)| = \frac{1}{\sqrt{1 - 2a\cos(\omega) + a^2}} \ \ \forall \ \ a \in \{-1, 1\} \tag{3}$$

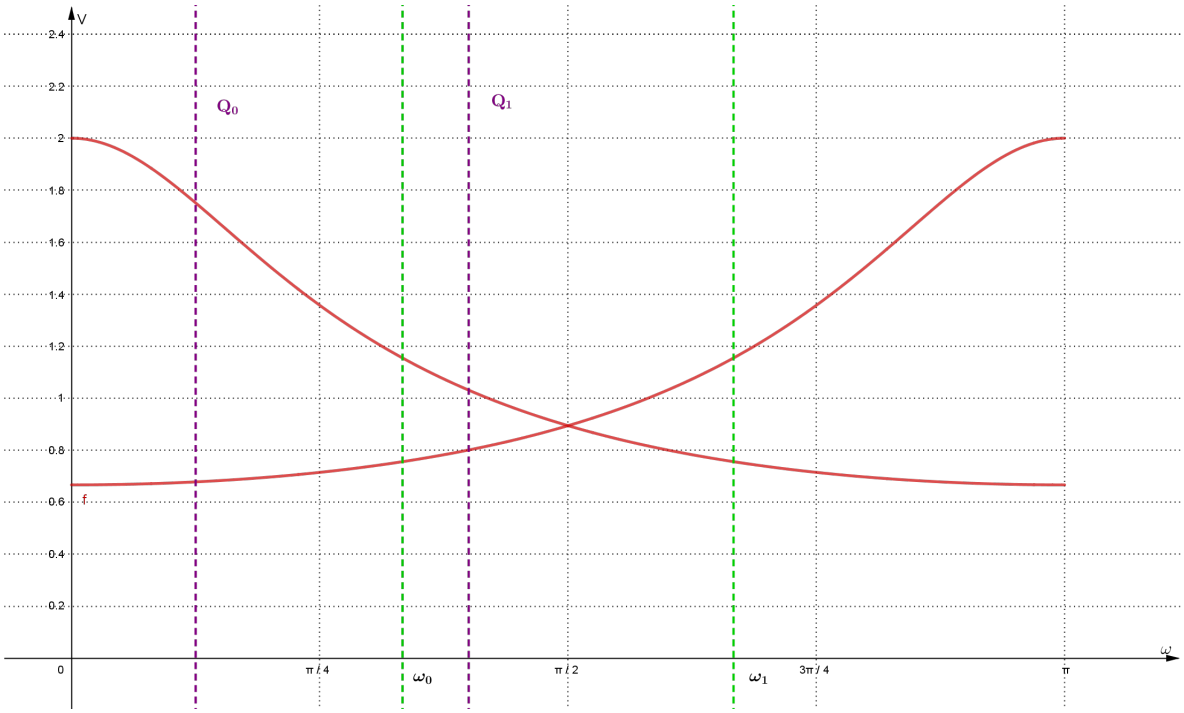
Ved og betrakte (3) for ulike verdier av a, observeres det at for positive verdier av a oppfører IIR filteret seg som et lavpassfilter, mens for negative verdier av a vil IIR filteret oppføre seg som et høypassfilter. Dette er illustrert i Figure 3.



Figur 3: Amplituderesponsen til IIR filter for ulike verider av a.

Ved og bestemme passende verdier for a slik at modulene H_1 og H_0 i Figure 2 oppfører seg som hhv. et høypass og lavpass- filter, kan middelverdiene sammenliknes for og avgjøre hvorvidt inngangssignalet har frekvensen f_0 eller f_1. Som Figure 3 viser, gir ingen av verdiene for a et filter, lavpass eller høypass, med et spesielt bredt passbånd. (Dette gjelder også for de verdier av a på intervallet $\in \{-1, 1\}$ som ikke er plottet i Figure 3). Signaler som inneholder f_0 eller f_1 vil nok undergå en betydelig grad av dempning. Dette er i midlertid helt greit ettersom det er differansen mellom amplitudene til $y_1[n]$ og $y_0[0]$ som skal brukes til og avgjøre. Det er altså bare nødvendig og bestemme a slik at differansen mellom $y_1[n]$ og $y_0[n]$ er "stor nok" til at den kan detekteres.

Det er også nødvendig og sørge for at lavpass og høypassfilterene er tilpasset f_0 og f_1. Observer fra Figure 3 at grafene til amplituderesponsen for ulike verdier av a møtes i $\omega = \frac{\pi}{2}$. For å kunne sammenlikne amplitudene til $y_1[n]$ og $y_0[n]$ er det ønskelig at $\omega = \frac{\pi}{2}$ ligger midt mellom $\omega_0 = 2\pi f_0$ og $\omega_1 = 2\pi f_1$, som illustrert i Figure 4. Dette oppnås ved å velge en samplingsfrekvens,f_s, slik at $\frac{f_s}{4} = \frac{f_0+f_1}{2}$. Dersom de lilla linjene, Q_0 og Q_1 i Figure 4 hadde representert frekvensene f_0 og f_1, ville sammenlikningen av amplituder vært ubrukelig.



Figur 4: Amplituderesponsen med en vilkårlig valgt verdi på a, ω_0 og ω_1. Det er ønskelig at f_s velges slik at $\frac{\pi}{2}$ havner midt mellom ω_0 og ω_1. Linjene Q_0 og Q_1 illustrerer et eksempel på uønsket plassering av frekvens.

3 Realisering og test

For å oppfylle kravet om et implementeringsareal på mindre enn 4cm^2 blir en ATmega382 microcontroller [5] montert på et Arduino UNO kort [2] anvendt. Arduino kortets innebygde A/D konverterer vil bli anvendt til samplingen av signalet. Ettersom implementerigen skal bli gjort på en microkontroller, er realiseringen av modulene diskutert i kapittel 2 nødt til å bli gjort digitalt ved hjelp av kode som lastes inn på microkontrolleren. Programmeringsspråket C vil bli anvendt til dette. Koden som ble skrevet og brukt under realisering og testing er inkludert i Vedlegg A. Den interesserte leser anbefales å lese gjennom dette for en full forståelse av hvordan FSK-demodulatoren er realisert.

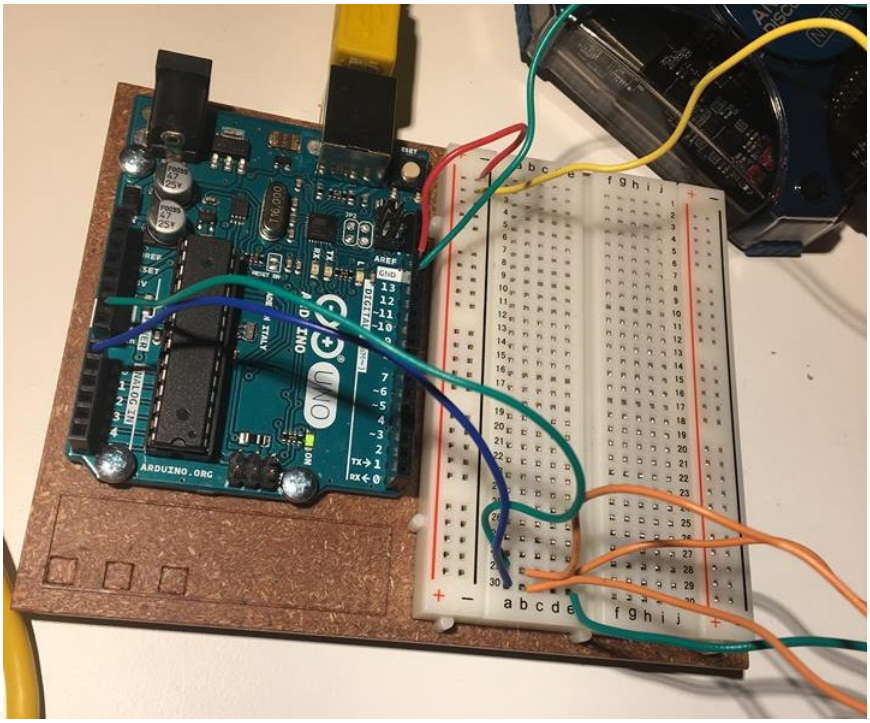
Signalet som vil bli brukt til å teste FSK-demodulatoren består av frekvensene $f_0 = 400Hz$ og $f_1 = 800Hz$. For at disse frekvensene skal ha samme avvik fra $\omega = 0$, som diskurtert i kapittel 2 og illustrert i Figure 4, vil en samplingsfrekvensen f_s være $f_s = 2400Hz$. Arduino kortets A/D koverterer kan bare sample signaler mellom 0V og 5V [3], og derfor vil en DC-offset på 2.5V bli lagt til, slik at samplingen blir korrekt. Verdien på multiplikasjonsfaktoren a, fra likning (1), blir valgt til $a = \pm 0.5$. Dette vil gi en tilstrekkelig differanse mellom signalet fra lavpassfilteret og høypassfilteret.

Under designet av koden måtte visse hensyn bli tatt, som ledet til noen avvik fra den prinsippielle løsningen. Det viktigste designhensynet som må tas, er at hele prosessen beskrevet i

kapittel 2.1 og illustrert i Figure 2 må fullføres før neste punktprøvningsverdi blir tatt. Ettersom punktprøvningsfrekvensen er $f_s = 2400Hz$ setter det et krav om at ting må skje raskt. Alle tidskrevende operasjoner bør følgelig prøves å unngå. For en prossesor er divisjon en tidskrevende operasjon. Det eneste stedet det blir utført divisjon i behandlignsprossesen, er når middelverdien skal beregnes. Men, denne divisjonen er egentlig helt irrelevant, ettersom middelverdiene bare blir sammenliknet med hverandre. Derfor kan divisjonen bli unngått og funksjonaliteten vil bli ivaretatt. Følgelig må terkselverdien som blir brukt til og bestemme $u(t)$ bli justert opp tilsvarende, ettersom den var basert på målinger av middelverdien. Et annet hensyn som blir tatt er at antall målinger som middelverdien (nå uten divisjon) blir basert på er relativt lav, slik at oppdatering av registeret ikke tar for langt tid.

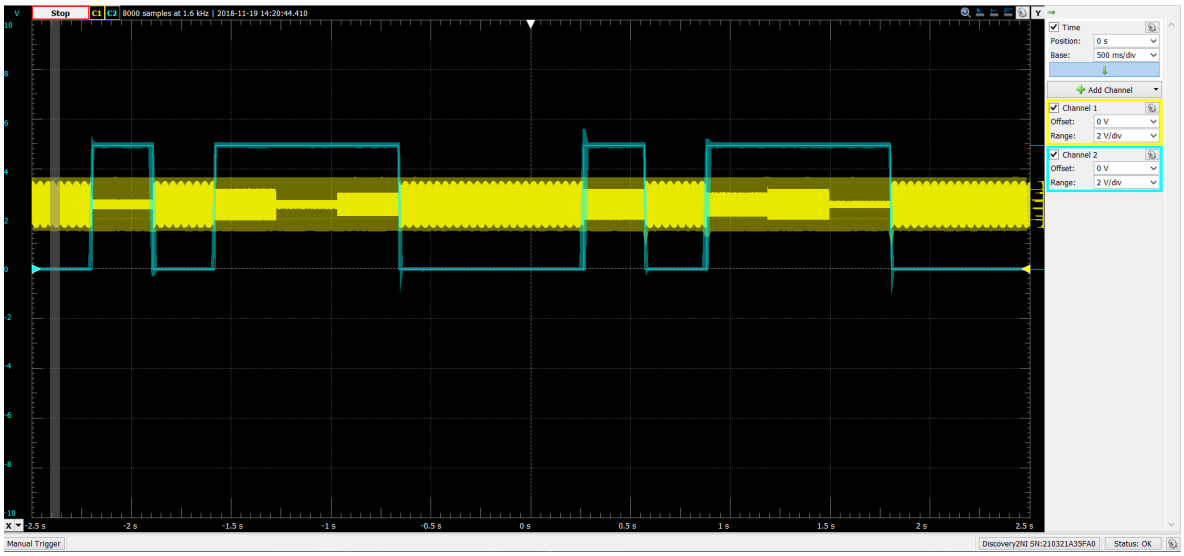
Arduinoen ble koblet til en signalgenerator som sendte testsignalet inn på kortet. Med et oscilloskop og programmet *Digilent Waveforms* [4] ble testsignalet, og utsignalene fra FSK-demodulatoren, $b(t)$ og $u(t)$, observert.

Den fysiske realiseringen av FSK-demodulatoren er vist i Figure 5.

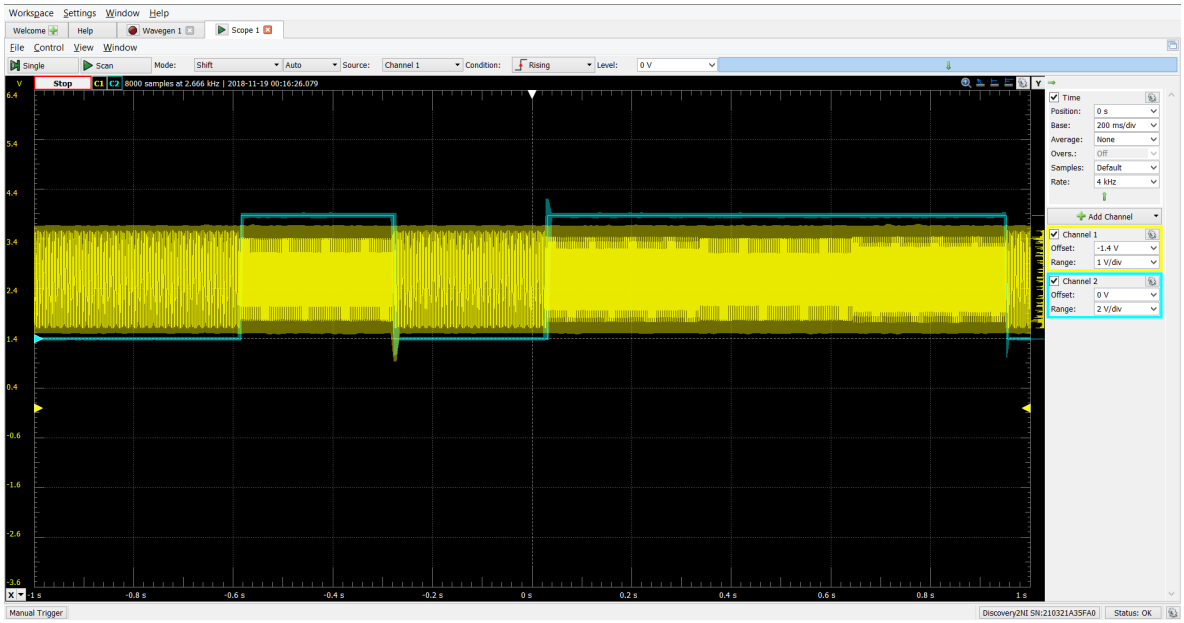


Figur 5: Den fysiske realiseringen av FSK-demodulatoren.

Figure 6 viser testsignalet i gult og utgangsignalet til FSK-modualtoren, $b(t)$, som skal representere bit-sekvensen i blått. Som forventet er $b(t)$ logisk `HØY` når testsignalet har frekvensen f_1. Tilsvarende er det logisk `LAV` når testsignalet har frekvensen f_0. Endrigen på logisk nivå i $b(t)$ er uten synlig forsinkelse. Figure 7 viser testsignal og $b(t)$ med en kortere steglengde på tidsaksen for og tydeligere vise frekvensforskjellene.

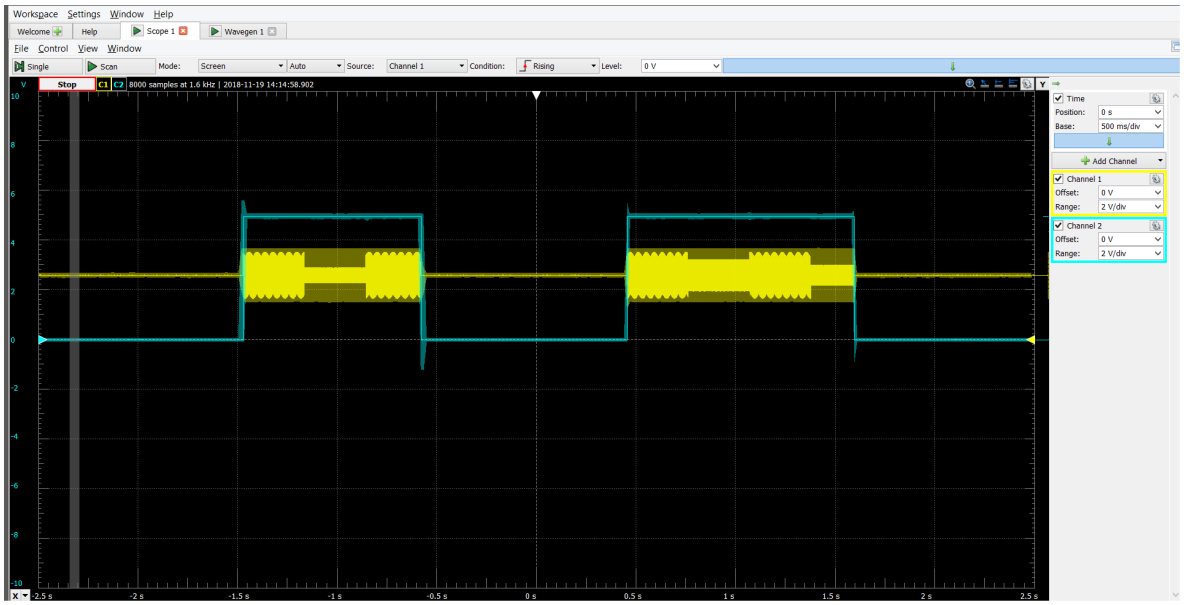


Figur 6: Testsignalet med frekvenskomponenter $f_0 = 400Hz$ og $f_1 = 800Hz$ (Gul). Utgangssignalet som skal representere bit-sekvensen $b(t)$ (Blå).

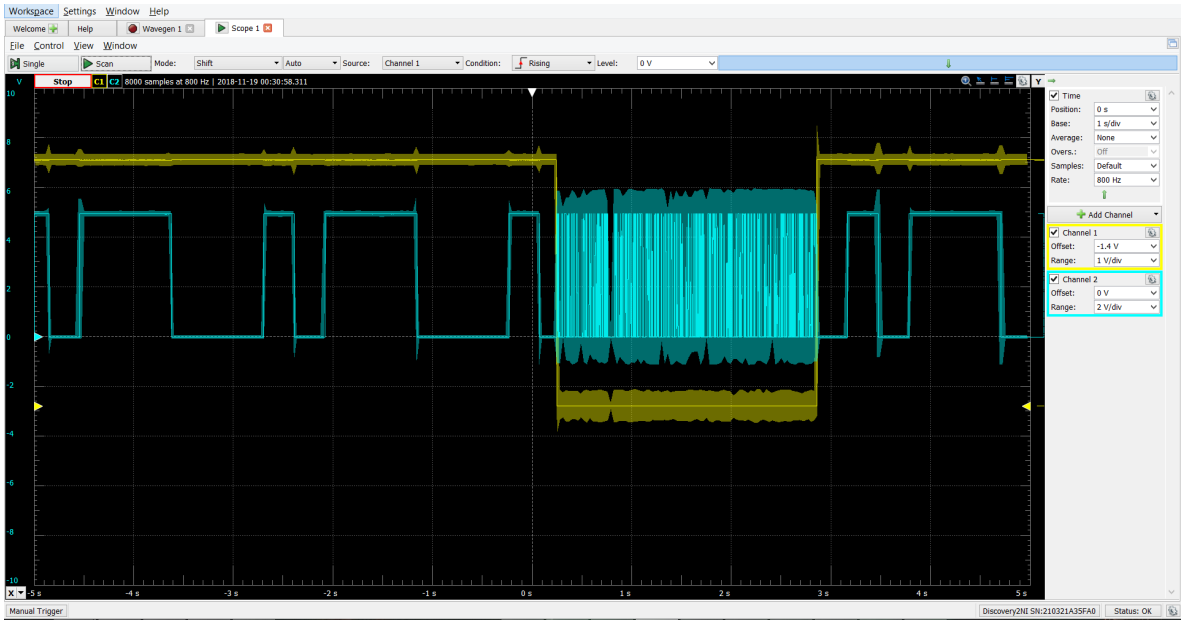


Figur 7: Testsignalet med frekvenskomponenter $f_0 = 400Hz$ og $f_1 = 800Hz$ (Gul). Utgangssignalet som skal representere bit-sekvensen $b(t)$ (Blå). Kortere steglengde på tidsaksen for og tydligere vise frekvensforskjellen.

Testingen av utgangssignalet $u(t)$ ble gjort ved og skru testsignalet av og på med gjevne mellomrom. Resultatene er gjengitt i Figure 8 og Figure 9.



Figur 8: Testsignalet som blir skrudd av og på (Gul). Utgangssignalet som skal overvåke inngangssigalet $u(t)$ (Blå).



Figur 9: Utgangssignalet som skal representere bitsekvensen $b(t)$ (blå). Utgangssignalet som skal overvåke inngangssigalet $u(t)$ (gul).

Testene viser at $u(t)$ blir logisk `LAV` når testsignalet blir skrudd av, akkuratt som forventet.

Det ble også testet hva som skjer med $u(t)$ når testsignalet var på, men med en annen frekvens enn f_0 og f_1. Da ble $u(t)$ konstant logisk `HØY`. Brukeren av FSK-demodulatoren bør være klare over dette. Enten bør en løsning på dette problemet implementeres, hvor $u(t)$ blir logisk `LAV`

ved andre frekvenser enn f_0 og f_1. Ellers bør mottaker forsikre seg om at signalet som blir sendt kun inneholder frekvenskomponentene f_0 og f_1. Sistnevnte virker ikke som et urimelig krav og stille til avsender.

4 Konklusjon

En FSK-demodulator har blitt designet og implementert med et areal mindre enn 4cm^2. Representasjonen av bit-sekvensen fungerer som ønsket. Signalovervåkeren kan detektere hvorvidt det er innkommende signal eller ikke, men kan ikke skille mellom ulike frekvenskomponenter. Brukeren bør være klar over dette og ta nødvendige hensyn deretter.

Referanser

[1] Lars Lundheim, *Digital kommunikasjon med FSK*, Elsys-2018-LL-1.2, Versjon 2, 2018.

[2] Arduino, *Arduino UNO Documentation*,
Hentet 19 november 2018.
https://store.arduino.cc/arduino-uno-rev3

[3] Arduino, *Arduino ReadAnalogVoltage*,
Hentet 19 november 2018.
https://www.arduino.cc/en/Tutorial/ReadAnalogVoltage

[4] Digilent *Waveforms software - Product description*
Hentet 10 Jan 2018.
https://store.digilentinc.com/waveforms-previously-waveforms-2015/

[5] Microchip, *ATmega328 Documentation*
Hentet 19 november 2018.
https://www.microchip.com/wwwproducts/en/ATmega328

A Kodeimplementasjonen

Listing 1: Implentasjonen av FSK-demodulator, med språket C.

```
1  // D8 - Kode til FSK Modulator. Basert p  utgitt eksempelkode.
2  // Jakob Vahlin - 18.11.2018
3
4  #include <TimerOne.h>
5  #include <math.h>
6
7  // Globale variaber
8  volatile int sample; // Holder siste sample
9  bool newSample; // Sjekker om ny sample er tatt
10 int static samplingsintervall = 417; // -> F_s = 2.4kHz.
11 float static a_0 = 0.5; // Multiplikator i LP
12 float static a_1 = -0.5; // Multiplikator i HP
13
14 // Lengden til registerert, dvs hvor mangle samples du baserer
      middelveri p . Boer holdes relativt lav, for kort kjoringstid.
15 #define avg_len 12
16
17 // Monitorerte middelverdi signalene og la terskelen godt under
      halvparten av den verdien
18 #define VALIDLIMIT 500
19
20 #define pinB 13 // Pin til B(t)
21 #define pinU 12 // Pin til U(t)
22
23 // Register for lagring av verdier til middelverdiberegning
24 volatile float avg_register_0[avg_len];
25 volatile float avg_register_1[avg_len];
26
27 volatile int arraycounter = 0; //
28
29 // "Minnet" til rekursivt filter. Maa vaere null ved forste
      sampling.
30 volatile float y1_nPrev = 0;
31 volatile float y0_nPrev = 0;
32
33 void setup() {
34    // Maa laste inn 0ere i middelverdi registeret fra starten.
35   for (int i = 0; i < avg_len; i++) {
36     avg_register_0[i] = 0;
37     avg_register_1[i] = 0;
38   }
39
40   // Oppsett av Pins
```

```
41    pinMode(pinB, OUTPUT); // B
42    pinMode(pinU, OUTPUT); // U
43    // Oppsett av timer interrupt
44    Timer1.initialize(samplingsintervall);
45
46    // Argumentet i "attachInterrupt" bestemmer hvilken funskjon som
          er interrupt handler
47    Timer1.attachInterrupt(takeSample);
48  }
49
50  // Interrupt-handler (denne kalles ved hvert interrupt)
51  // Dette er lesning av A/D converter.
52  void takeSample(void) {
53    sample = analogRead(0); // Sampler p   A0
54    newSample = true;
55  }
56
57  void loop() {
58    float y1_n;
59    float y0_n;
60    float fsample;
61    int i; // Til for-lokke
62
63    float y_avg_1;
64    float y_avg_0;
65    if (newSample) {
66      // Ny sample er tatt
67      // Implementerer foerst HP og LP filterene.
68
69      // Gjor sample om til float.
70      // Naar den samplet et "tomt" signal, ble verdien 532.
          Subtraherer for og nullstille.
71      fsample = (float)(sample - 532);
72
73      // Modul H1 - HP
74      y1_n = fsample + (a_1 * y1_nPrev);
75      y1_nPrev = y1_n;
76
77      // Modul H0 - LP
78      y0_n = fsample + (a_0 * y0_nPrev);
79      y0_nPrev = y0_n;
80
81      // Neste steg er og ta abs.verdi av begge
82      y1_n = fabs(y1_n);
83      y0_n = fabs(y0_n);
84
```

```
85      // Maa legge abs.verdi inn i register, slik at middelverdi kan
            beregnes.
86      avg_register_0[arraycounter] = y0_n;
87      avg_register_1[arraycounter] = y1_n;
88
89      // Dersom programmet gaar "uendelig lenge" vil arraycounter
            nullstilles naar verdien kommer utenfor bit-range, derfor
            modulo. Slik holdes registeret oppdatert.
90      arraycounter++;
91      arraycounter = arraycounter % avg_len;
92
93      // Neste steg er    beregne middelveriden av avg_len samples.
94      // Kort forlokke slik at programmet ikke bruker lang tid paa en
             sample.
95      // M  holde avg_len forholdsvis liten.
96
97      // Fra modul H1
98      y_avg_1 = 0;
99      for (i = 0; i < avg_len; i++) {
100       y_avg_1 += avg_register_1[i];
101     }
102     // Trenger ikke aa egentlig beregne middelveriden, fordi den
            brukes til og sammenlikne H1 og H0.
103     // Divisjon tar mye tid, vil unngaa det.
104
105     //Modul H0
106     y_avg_0 = 0;
107     for (i = 0; i < avg_len; i++) {
108       y_avg_0 += avg_register_0[i];
109     }
110     // Beregner ikke egentlig middelveriden. Se kommentar over.
111
112     // Neste steg er    sammenlikne veridene. H1 > H0 -> B = HIGH,
            H0 > H1 -> B = LOW
113     if (y_avg_1 > y_avg_0) {
114       // High paa B
115       digitalWrite(pinB, HIGH);
116
117     }
118     else {
119         // LOW paa B
120       digitalWrite(pinB, LOW);
121
122     }
123
124    // M  ogs  sjekke om signalet er gyldig
125    if (y_avg_0 > VALIDLIMIT || y_avg_1 > VALIDLIMIT) {
```

```
126         // HIGH paa U
127         digitalWrite(pinU, HIGH);
128 
129       }
130       else {
131         // LOW paa U
132         digitalWrite(pinU, LOW);
133       }
134 
135       newSample = false;
136     }
137 }
```