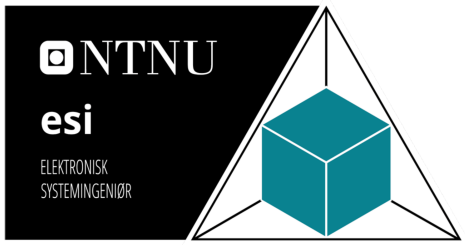
Designnotat 10



Tittel: FSK Demodulator

Forfatter: Temmam Nabil Altaie

Versjon: 2.0 | **Dato: 05.06.2026**

Innhold

1 Problembeskrivelse **2**

2 Prinsipiell løsning **3**

2.1 Overordnet systemidé . . . 3
2.2 Funksjonell oppdeling og signalflyt . . . 3
2.3 Frekvensselektiv filtrering . . . 3
2.4 Amplitude- og energiestimering . . . 4
2.5 Bitbeslutning og timing . . . 4
2.6 Begrunnelse for valgt prinsipp . . . 4
2.7 Oppsummering av sentrale prinsipper . . . 4

3 Realisering og test **5**

3.1 Sampling og sanntidsstruktur . . . 5
3.2 Frekvensselektiv filtrering . . . 6
3.3 Energideteksjon og bitbeslutning . . . 6
3.4 Fastpunktimplementasjon . . . 6
3.5 MATLAB-verifikasjon . . . 7
3.6 Eksperimentell verifikasjon . . . 8
3.7 Oppsummering av test . . . 9

4 Konklusjon **10**

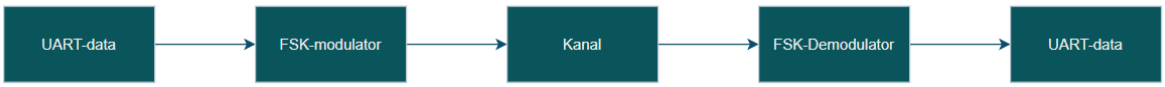
5 Takk **10**

6 Kodevedlegg **11**

6.1 Matlab.m . . . 11
6.2 main.c . . . 12

1 Problembeskrivelse

UART (Universal Asynchronous Receiver/Transmitter) er en utbredt metode for seriell digital kommunikasjon mellom elektroniske systemer. Tradisjonelt forutsetter UART en direkte elektrisk forbindelse mellom sender og mottaker, bestående av signalledninger og felles referansejord. I dette designprosjektet erstattes den fysiske forbindelsen med en trådløs overføring basert på frekvensskiftmodulasjon (FSK).



Figur 1: Overordnet system for UART-kommunikasjon via FSK-modulasjon. Demodulatoren er delen som designes i dette prosjektet.

Ved FSK representeres digitale bitverdier ved to ulike sinusfrekvenser. I det gitte testsignalet benyttes frekvensene 300 Hz og 200 Hz, der én frekvens representerer logisk 0 og den andre representerer logisk 1. Bitperioden er 50 ms, som tilsvarer en bitrate på $20\,\mathrm{bit\,s^{-1}}$.

Problemstillingen i prosjektet er å designe og implementere en digital FSK-demodulator som kan motta et analogt FSK-modulert signal og rekonstruere den opprinnelige binære bitsekvensen. Den rekonstruerte bitstrømmen skal kunne behandles videre som et UART-signal og grupperes i bytes.

Målet med prosjektet er dermed å:

- detektere hvilken av de to frekvensene som er til stede i hvert bitintervall,
- generere korrekt binærverdi,
- rekonstruere bytes fra den mottatte bitstrømmen.

Prosjektet omfatter kun demodulatoren. Radiomottaker og modulasjon på sendersiden er ikke en del av arbeidet. Fokus er derfor på digital signalbehandling og frekvensdeteksjon av det mottatte analoge signalet.

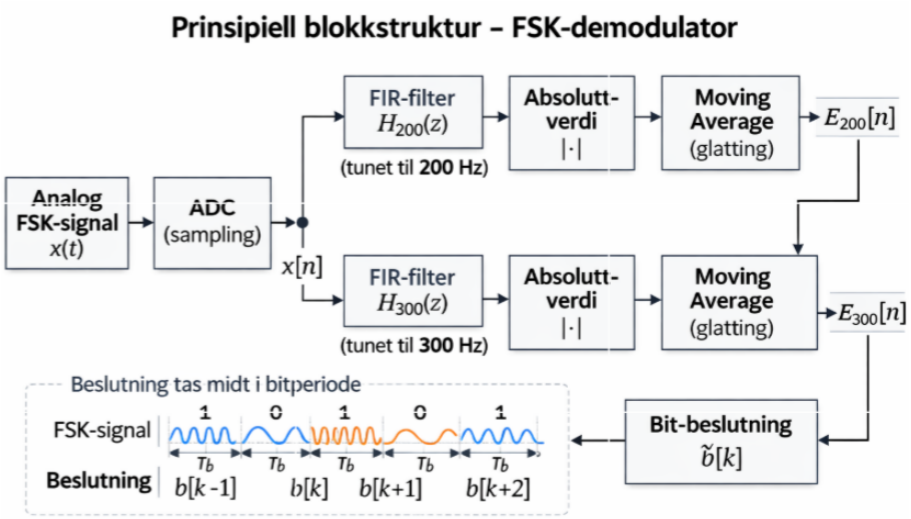
2 Prinsipiell løsning

2.1 Overordnet systemidé

FSK-demodulering kan utføres ved å måle hvilken av to definerte frekvenser som dominerer i det mottatte signalet innenfor hvert bitintervall. I DP10 er de to frekvensene 200 Hz og 300 Hz, som representerer henholdsvis én og null (eller omvendt avhengig av definisjon på sendersiden). Hovedideen i løsningen er derfor å estimere energien i signalet rundt hver av disse frekvensene og ta en binær beslutning basert på hvilken energi som er størst.

Løsningen bygges opp som en digital signalkjede der det analoge FSK-signalet først samples, deretter filtreres i to parallelle grener (én per frekvens), før det utføres en enkel amplitudeevaluering og beslutningslogikk. Bitbeslutningen tas på et tidspunkt midt i bitperioden for å redusere påvirkning fra transienter i overgangen mellom to påfølgende bits.

2.2 Funksjonell oppdeling og signalflyt



Figur 2: Prinsipiell blokkstruktur for FSK-demodulator. Det mottatte signalet samples, filtreres i to parallelle grener tunet til henholdsvis 200 Hz og 300 Hz, før energisammenligning gir binær beslutning.

Figur 2 viser en overordnet blokkstruktur for demodulatoren. Den mottatte signalformen samples med ADC og behandles videre som en diskret tidssekvens $x[n]$. Denne sekvensen filtreres i to parallelle grener med hvert sitt frekvensselektive filter. Filterutgangene transformeres deretter til et mer stabilt energimål ved å benytte absoluttverdi og glatting (moving average). Til slutt sammenlignes de to energimålene for å bestemme bitverdien $\hat{b}[k]$ for bitindeks k.

2.3 Frekvensselektiv filtrering

For å skille mellom de to FSK-frekvensene benyttes to frekvensselektive filtre:

- $H_{200}(z)$: gir høy respons ved 200 Hz
- $H_{300}(z)$: gir høy respons ved 300 Hz

Ved innsignal dominert av 200 Hz forventes det at utgangen fra $H_{200}(z)$ blir betydelig større enn utgangen fra $H_{300}(z)$, og motsatt når innsignalet domineres av 300 Hz. Filterutgangene kan skrives som

$$y_{200}[n] = (h_{200} * x)[n], \tag{1}$$
$$y_{300}[n] = (h_{300} * x)[n], \tag{2}$$

der $h_{200}[n]$ og $h_{300}[n]$ er impulsresponsene til de to filtrene, og $*$ betegner konvolusjon.

2.4 Amplitude- og energiestimering

Filterutgangene er oscillerende signaler, og direkte sammenligning av $y_{200}[n]$ og $y_{300}[n]$ vil være sensitivt for fase og nullkryssinger. For å oppnå et mer robust mål på "styrken" til hver gren benyttes derfor absoluttverdi etterfulgt av glatting med et moving-average-filter. Et generelt energimål kan uttrykkes som

$$E_f[n] = \frac{1}{L} \sum_{i=0}^{L-1} |y_f[n-i]|, \quad f \in \{200, 300\}, \tag{3}$$

der L er lengden på glattingen (antall samples). Dette gir et saktevarierende mål som i større grad reflekterer hvilken frekvens som er til stede i gjeldende bitintervall, og reduserer påvirkning fra støy og kortvarige variasjoner.

2.5 Bitbeslutning og timing

Bitbeslutningen tas ved å sammenligne de to energimålene. Dersom energien i 200 Hz-grenen er større enn energien i 300 Hz-grenen, tolkes biten som 1, ellers som 0:

$$\hat{b}[k] = \begin{cases} 1, & E_{200}[n_k] > E_{300}[n_k] \\ 0, & \text{ellers} \end{cases} \tag{4}$$

der n_k er sampleindeksen som representerer beslutningstidspunktet for bit k. For å redusere feiltolkning ved bitoverganger velges n_k typisk midt i bitperioden. I et FSK-signal kan det forekomme transienter når frekvensen skifter mellom to bits, og midtsampling gir derfor en mer stabil beslutning sammenlignet med sampling nær bitgrensene.

2.6 Begrunnelse for valgt prinsipp

Valgt demoduleringsprinsipp basert på to parallelle frekvensselektive filtre gir en intuitiv og implementerbar løsning på mikrokontroller. Metoden er modulær og robust: filtre kan justeres ved behov, og beslutningslogikken reduseres til en enkel sammenligning. Samtidig utnyttes grunnleggende DSP-prinsipper (frekvensselektivitet, envelopdeteksjon og tidsdiskret beslutning) som er direkte relevante for prosjektets læringsmål.

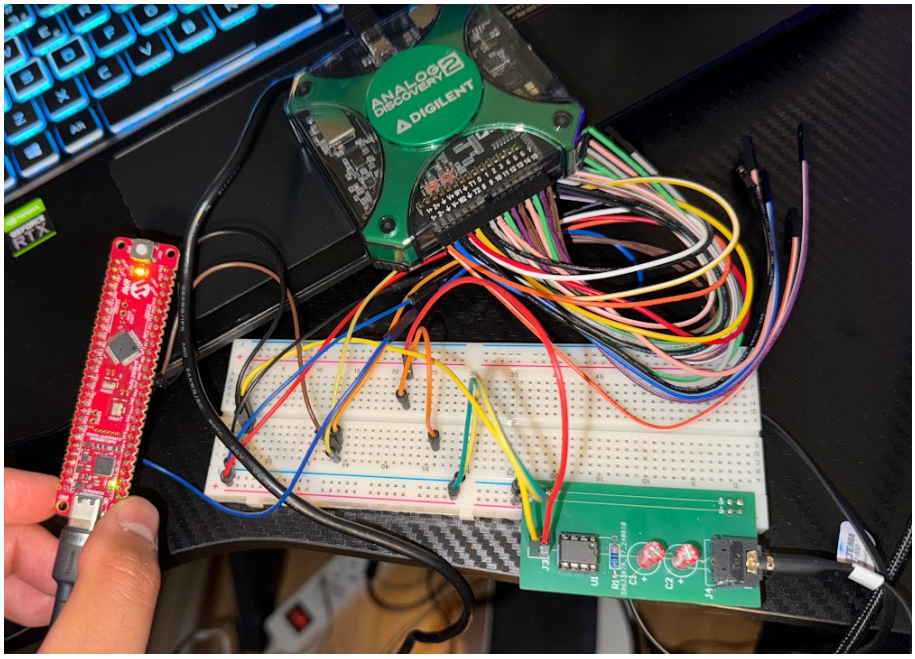
2.7 Oppsummering av sentrale prinsipper

Den prinsipielle løsningen bygger på:

- Diskret sampling av analogt FSK-signal ($x(t) \rightarrow x[n]$)
- Frekvensselektiv filtrering for å separere komponenter ved 200 Hz og 300 Hz
- Envelopdeteksjon gjennom absoluttverdi og glatting (moving average)
- Energisammenligning og terskeldeteksjon for binær beslutning
- Bit-timing med beslutning midt i bitperioden for økt robusthet

3 Realisering og test

FSK-demodulatoren er implementert som et sanntidssystem på mikrokontrolleren AVR128DB48. Løsningen er avbruddsdrevet og basert på fastpunktberegninger (Q15) for å sikre deterministisk kjøretid og lav ressursbruk.



Figur 3: Kretsen som ble oppkoblet. Analog discovery 2, AVR128db48 og lyttekort for å tolke signalene ved audio.

3.1 Sampling og sanntidsstruktur

Det analoge FSK-signalet samples ved hjelp av ADC med samplingsfrekvens 1000 Hz. Med en bitperiode på 50 ms gir dette 50 samples per bit, noe som er tilstrekkelig for frekvensdeteksjon av 200 Hz og 300 Hz.

En timer genererer periodiske avbrudd som utfører følgende operasjoner:

- Start av ADC-konvertering
- Oppdatering av ringbuffer
- FIR-beregning for begge frekvensgrener

- Energiberegning og moving average

ADC-verdien forskyves til nullnivå ved å trekke fra midtverdien (512 for 10-bit ADC).

3.2 Frekvensselektiv filtrering

To lineære FIR-filtre benyttes for å estimere energien rundt henholdsvis 200 Hz og 300 Hz. Filtrene er implementert som konvolusjon:

$$y[n] = \sum_{k=0}^{N-1} b[k]\, x[n-k]$$

Koeffisientene er generert eksternt og skalert til Q15-format:

$$b_i^{(Q15)} = \text{round}(32768 \cdot b_i)$$

Akkumulering skjer i 32-bit heltall før høyreskift på 15 bit.

3.3 Energideteksjon og bitbeslutning

Filterutgangen rettes opp ved absoluttverdi:

$$e[n] = |y[n]|$$

Videre glattes signalet med et moving average-filter:

$$E[n] = \frac{1}{2^M} \sum_{k=0}^{2^M-1} e[n-k]$$

Bitbeslutningen tas midt i hver bitperiode (etter 25 samples):

$$\hat{b}[k] = \begin{cases} 1 & E_{200} > E_{300} \\ 0 & \text{ellers} \end{cases}$$

Bits grupperes i bytes (LSB først). Synkronisering oppnås ved deteksjon av tre påfølgende preamble-bytes (0x55).

3.4 Fastpunktimplementasjon

For effektiv kjøring benyttes:

- Q15-koeffisienter
- 16-bit signalbuffer

- 32-bit FIR-akkumulator
- Ringbuffer for kontinuerlig sampling

Dette muliggjør stabil sanntidsprosessering ved 1000 Hz.

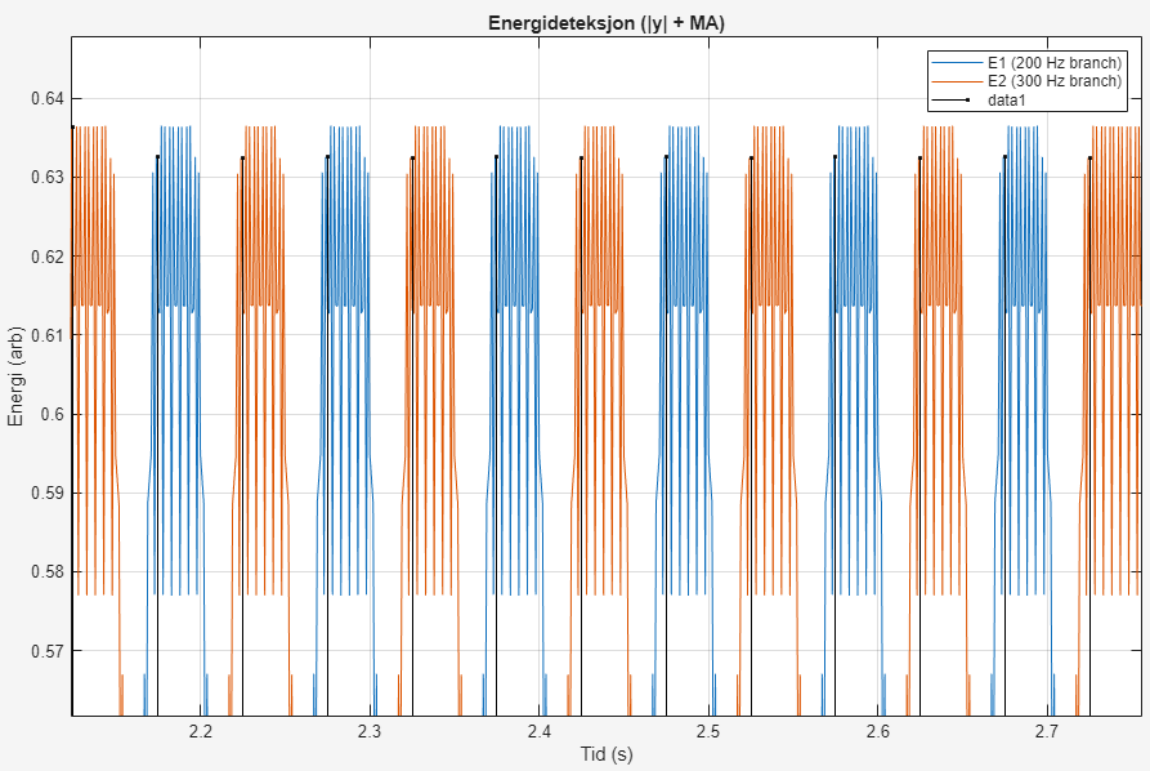
Test og verifikasjon

Formålet med testfasen var å verifisere korrekt frekvensdeteksjon, stabil energisammenligning og korrekt byte-rekonstruksjon.

3.5 MATLAB-verifikasjon

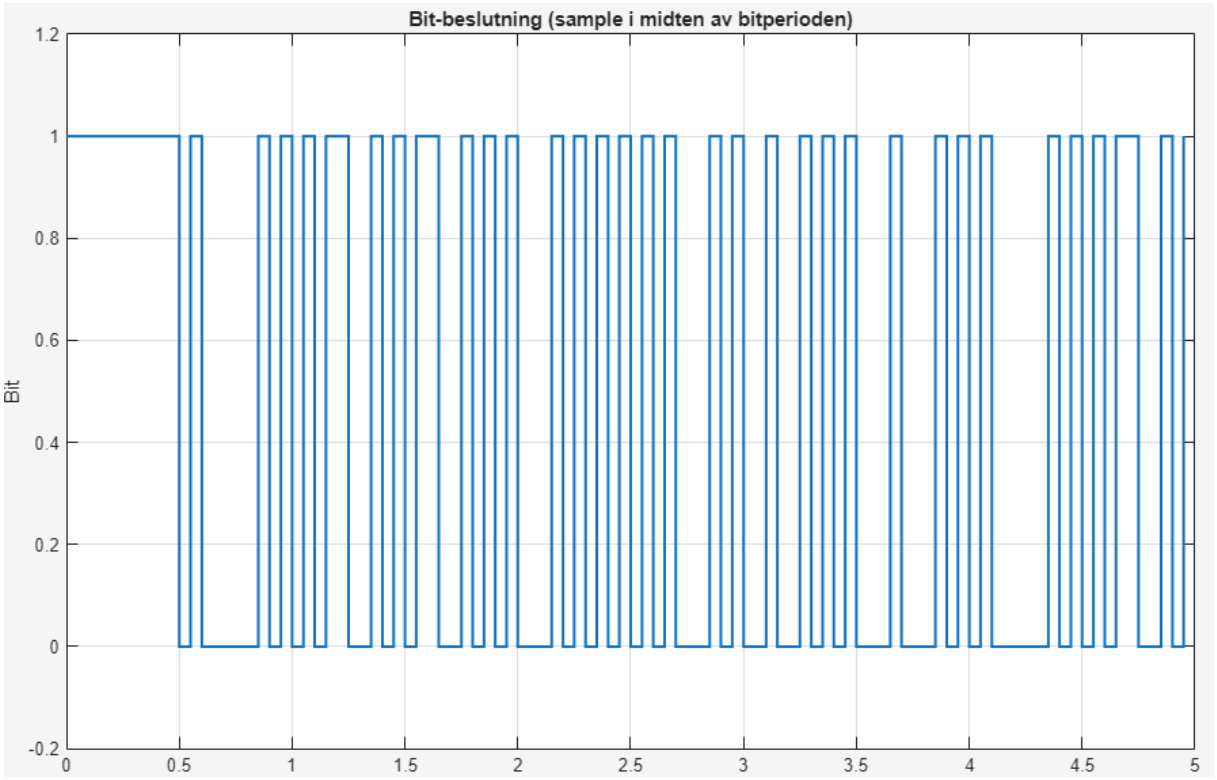
Før implementasjon på mikrokontroller ble algoritmen verifisert i MATLAB.

Figur 4 viser energikurvene etter absoluttverdi og moving average for et zoomet tidsintervall. Det observeres tydelig dominans i korrekt filter innen hver bitperiode. Kryssing mellom energikurvene skjer kun ved bitoverganger.



Figur 4: Energideteksjon for 200 Hz og 300 Hz etter absoluttverdi og moving average. Korrekt filter dominerer innen hver bitperiode.

Figur 5 viser bitbeslutningen tatt midt i hver bitperiode. Signalet er stabilt og endres kun ved faktiske bitgrenser.



Figur 5: Bitbeslutning i MATLAB basert på energisammenligning.

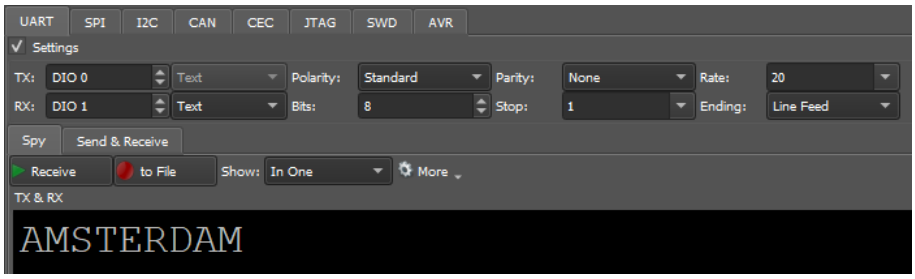
Disse resultatene bekrefter at algoritmen fungerer teoretisk korrekt.

3.6 Eksperimentell verifikasjon

Testsignal ble generert med AD2 (Analog Discovery 2) med frekvensene 200 Hz og 300 Hz og bitperiode 50 ms. Signalet ble koblet til ADC-inngangen på AVR128DB48.

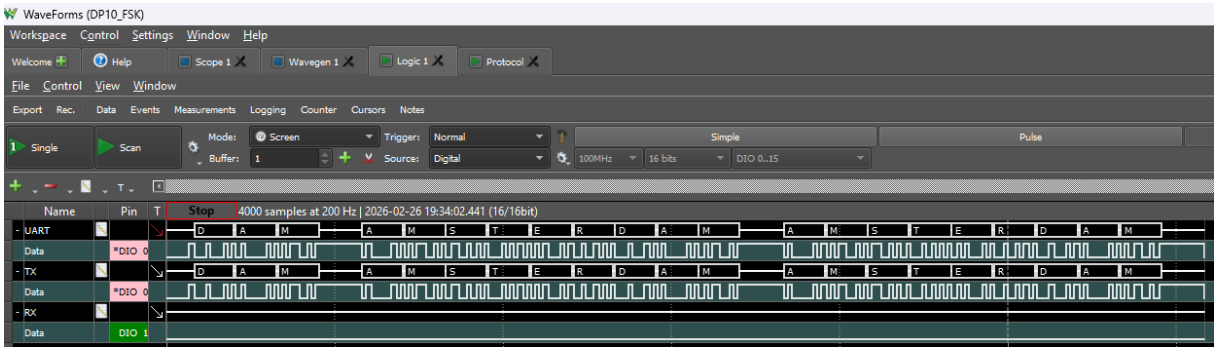
UART-utgangen ble analysert både via AD2 og ekstern logic analyzer.

Figur 6 viser korrekt rekonstruert tekst via AD2-protokollanalyse.



Figur 6: UART-dekoding via AD2 som viser korrekt rekonstruert tekst.

Figur 7 viser tilsvarende dekoding via logic analyzer med korrekt baudrate-innstilling.



Figur 7: UART-analyse med logic analyzer. Korrekt bit rate gir riktig dekoding.

Den mottatte teksten ble korrekt rekonstruert som:

```
AMSTERDAM
```

Dette bekrefter at hele signalbehandlingskjeden – fra ADC-sampling via filtrering og energideteksjon til byte-rekonstruksjon – fungerer korrekt.

3.7 Oppsummering av test

Følgende ble verifisert:

- Korrekt frekvensdeteksjon per bit
- Stabil energiforskjell mellom filtrene
- Robust midtperiode-sampling
- Riktig byte-rekonstruksjon

Systemet oppførte seg deterministisk uten sample-tap eller timingfeil.

4 Konklusjon

I dette prosjektet er det utviklet en digital FSK-demodulator for mottak av UART-data over en frekvensskiftmodulert kanal. Løsningen er basert på frekvensselektiv filtrering ved hjelp av to parallelle FIR-filtre, etterfulgt av energiestimering og komparatorbasert bitbeslutning.

Den prinsippielle strukturen ble implementert som et sanntidssystem på AVR128DB48 med avbruddsdrevet sampling og fastpunktberegninger. Q15-representasjon av filterkoeffisienter og bruk av 32-bit akkumulering sikrer numerisk stabilitet og effektiv ressursbruk. Systemet er dimensjonert for en punktprøvingsfrekvens på 1000 Hz, som gir tilstrekkelig oppløsning for å skille mellom 200 Hz og 300 Hz innenfor hver bitperiode.

Den teoretiske gjennomgangen viser at energien i det korrekt tunede filteret vil dominere innen hvert bitintervall, og at beslutning tatt midt i perioden gir robust frekvensdeteksjon. Byterekonstruksjon og synkronisering via preamble er implementert for å muliggjøre videre UART-behandling.

Videre arbeid består i full eksperimentell verifikasjon med fysisk oppsett, inkludert generering av FSK-signal via AD2 og observasjon av filterrespons og bitbeslutning i sanntid. Dette vil gi en komplett validering av systemets funksjon under reelle forhold.

Samlet sett demonstrerer prosjektet hvordan digital signalbehandling kan anvendes til robust frekvensdeteksjon og digital kommunikasjon over en analog kanal, samt hvordan en teoretisk modell kan realiseres effektivt på mikrokontroller.

5 Takk

Takk til faglærer.

6 Kodevedlegg

6.1 Matlab.m

```
%% Offline verifikasjon av FSK-demod (matcher AVR-oppsettet)
clear; close all; clc

[x, Fs] = audioread('FSK_1.wav');
x = x(:,1);

% Hvis WAV ikke er 1 kHz: resample til 1 kHz (du bruker 1k i koden)
Fs_target = 1000;
if Fs ~= Fs_target
    x = resample(x, Fs_target, Fs);
    Fs = Fs_target;
end

% Sentrer rundt 0 (AVR gj r adc-512)
x = x - mean(x);

% FIR taps (N=21) fra koden din
bf1 = [ ...
  0.0138  0.0055 -0.0240 -0.0390  0.0222 ...
  0.0980  0.0385 -0.1203 -0.1358  0.0557 ...
  0.1846  0.0557 -0.1358 -0.1203  0.0385 ...
  0.0980  0.0222 -0.0390 -0.0240  0.0055 ...
  0.0138];

bf2 = [ ...
  0.0138 -0.0055 -0.0240  0.0390  0.0222 ...
 -0.0980  0.0385  0.1203 -0.1358 -0.0557 ...
  0.1846 -0.0557 -0.1358  0.1203  0.0385 ...
 -0.0980  0.0222  0.0390 -0.0240 -0.0055 ...
  0.0138];

% Filtrering
y1 = filter(bf1, 1, x);
y2 = filter(bf2, 1, x);

% Energi: abs + moving average med L=16 (MA_N=4)
L = 16;
E1 = movmean(abs(y1), L);
E2 = movmean(abs(y2), L);

% Bit-sampling: 50 ms bitperiode -> 50 samples ved 1 kHz
Tb_samp = 50;
mid = 25; % sample_cnt==25
idx = mid:Tb_samp:length(E1);
bits = E1(idx) > E2(idx);   % eventuelt inverter hvis mapping er motsatt
bits = bits(:);

% Plot: energi og samplingpunkter
t = (0:length(x)-1)/Fs;

figure;
plot(t, E1, t, E2); grid on
legend('E1 (200 Hz branch)','E2 (300 Hz branch)')
xlabel('Tid (s)'); ylabel('Energi (arb)')
title('Energideteksjon (|y| + MA)')
hold on
stem(idx/Fs, max([E1(idx) E2(idx)],[],2), 'k.')
```

```
58
59 % Plot: bits
60 figure;
61 stairs((0:length(bits)-1)*0.05, bits, 'LineWidth', 1.5); grid on
62 xlabel('Tid (s)'); ylabel('Bit'); ylim([-0.2 1.2])
63 title('Bit-beslutning (sample i midten av bitperioden)')
64
65 % Byte building (LSB first) og finn preamble 0x55
66 bytes = [];
67 for k = 1:floor(length(bits)/8)
68     b = 0;
69     for i = 0:7
70         b = b + (bits((k-1)*8 + i + 1) * 2^i);
71     end
72     bytes(end+1) = b; %#ok<AGROW>
73 end
74
75 fprintf('F rste 40 bytes (hex):\n')
76 fprintf('%02X ', bytes(1:min(40,end)));
77 fprintf('\n')
78
79 % S k etter tre 0x55 p   rad
80 pat = [hex2dec('55') hex2dec('55') hex2dec('55')];
81 hit = strfind(bytes, pat);
82 if ~isempty(hit)
83     fprintf('Fant preamble 0x55x3 ved byteindeks %d\n', hit(1));
84 else
85     fprintf('Fant ikke preamble. Pr v    invertere bits.\n');
86 end
```

Kodeutdrag 1: Matlab kode for visualisering og analyse

6.2 main.c

```
1  /*
2   * AVR128DB48 - DP10 FSK demodulator (ADC -> 2x FIR -> |.| -> MA -> compare ->
       bit)
3   *
4   * PINOUT:
5   *   - PD1 : ADC input (AIN1)
6   *   - PD6 : DAC output
7   */
8
9
10
11 #include <avr/io.h>
12 #include <avr/interrupt.h>
13 #include <avr/cpufunc.h>
14 #include <stdint.h>
15 #include <stdbool.h>
16 #include <stdio.h>
17
18 #include "avr_driver.h"   // Must provide USART3_init() + printf redirection
19
20 /* ======================
21  * CONFIG
22  * ====================== */
23
24 #define SAMPLING_FREQUENCY 1000UL   // Hz
25 #define N                  21       // FIR taps
26
```

```
#define MA_N                4
#define MA_L                (1u << MA_N)   // 16

#define TIMER_TOP           ((F_CPU / SAMPLING_FREQUENCY) - 1)

/* ======================
 * SHARED (ISR <-> main)
 * ====================== */
static volatile uint8_t g_bit_out   = 0;
static volatile bool    g_bit_ready = false;

/* ======================
 * FILTER COEFFS (float)
 * ====================== */

static const float bf1[N] = {
    0.0138f,  0.0055f, -0.0240f, -0.0390f,  0.0222f,
    0.0980f,  0.0385f, -0.1203f, -0.1358f,  0.0557f,
    0.1846f,  0.0557f, -0.1358f, -0.1203f,  0.0385f,
    0.0980f,  0.0222f, -0.0390f, -0.0240f,  0.0055f,
    0.0138f
};

static const float bf2[N] = {
    0.0138f, -0.0055f, -0.0240f,  0.0390f,  0.0222f,
   -0.0980f,  0.0385f,  0.1203f, -0.1358f, -0.0557f,
    0.1846f, -0.0557f, -0.1358f,  0.1203f,  0.0385f,
   -0.0980f,  0.0222f,  0.0390f, -0.0240f, -0.0055f,
    0.0138f
};

/* Q15 coeff buffers */
static int16_t b1[N];
static int16_t b2[N];

/* ======================
 * DATA STRUCTURES
 * ====================== */

typedef struct {
    int16_t buffer[N];
    uint8_t writeIndex;
} ringbuffer_t;

typedef struct {
    uint16_t buf[MA_L];
    uint8_t  idx;
    uint32_t sum;
} ma_t;

static ringbuffer_t fir1_buffer = {0};   // intended: 200 Hz branch
static ringbuffer_t fir2_buffer = {0};   // intended: 300 Hz branch

static ma_t ma1 = {0};   // for |y1|
static ma_t ma2 = {0};   // for |y2|

/* ======================
 * INIT
 * ====================== */

static void clock_init(void)
{
```

```
89      // Set system clock to 24 MHz
90      ccp_write_io((void*)&CLKCTRL.OSCHFCTRLA, CLKCTRL_FRQSEL_24M_gc);
91  }
92
93  static void adc_init(void)
94  {
95      // PD1 (AIN1) as analog input
96      PORTD.PIN1CTRL &= ~PORT_ISC_gm;
97      PORTD.PIN1CTRL |= PORT_ISC_INPUT_DISABLE_gc;
98      PORTD.PIN1CTRL &= ~PORT_PULLUPEN_bm;
99
100     VREF.ADC0REF = VREF_REFSEL_VDD_gc;
101
102     ADC0.CTRLC  = ADC_PRESC_DIV12_gc;      // ADC clock ~2 MHz
103     ADC0.MUXPOS = ADC_MUXPOS_AIN1_gc;
104     ADC0.CTRLA  = ADC_ENABLE_bm | ADC_RESSEL_10BIT_gc;
105 }
106
107 static void dac_init(void)
108 {
109     // PD6 as DAC output
110     PORTD.PIN6CTRL &= ~PORT_ISC_gm;
111     PORTD.PIN6CTRL |= PORT_ISC_INPUT_DISABLE_gc;
112     PORTD.PIN6CTRL &= ~PORT_PULLUPEN_bm;
113
114     VREF.DAC0REF = VREF_REFSEL_VDD_gc;
115     DAC0.CTRLA   = DAC_ENABLE_bm | DAC_OUTEN_bm;
116 }
117
118 static void timer_init(void)
119 {
120     TCA0.SINGLE.PER     = (uint16_t)TIMER_TOP;
121     TCA0.SINGLE.INTCTRL = TCA_SINGLE_OVF_bm;
122     TCA0.SINGLE.CTRLA   = TCA_SINGLE_CLKSEL_DIV1_gc | TCA_SINGLE_ENABLE_bm;
123 }
124
125 /* ======================
126  * DSP HELPERS
127  * ====================== */
128
129 static inline void buffer_push(ringbuffer_t *rb, int16_t sample)
130 {
131     rb->buffer[rb->writeIndex] = sample;
132     rb->writeIndex++;
133     if (rb->writeIndex >= N) rb->writeIndex = 0;
134 }
135
136 static inline uint16_t adc_get_sample(void)
137 {
138     ADC0.COMMAND = ADC_STCONV_bm;
139     while (!(ADC0.INTFLAGS & ADC_RESRDY_bm)) {;}
140     ADC0.INTFLAGS = ADC_RESRDY_bm;
141     return ADC0.RES; // 10-bit right adjusted
142 }
143
144 static inline uint16_t abs16(int16_t v)
145 {
146     return (v < 0) ? (uint16_t)(-v) : (uint16_t)v;
147 }
148
149 static inline int16_t compute_fir(const ringbuffer_t *rb, const int16_t *
    coeffs_q15)
```

```
{
    int32_t acc = 0;
    int8_t read_idx = (int8_t)rb->writeIndex - 1;

    for (uint8_t i = 0; i < N; i++) {
        if (read_idx < 0) read_idx = (int8_t)(N - 1);
        acc += (int32_t)coeffs_q15[i] * (int32_t)rb->buffer[read_idx];
        read_idx--;
    }

    // Q15 * Q0 => Q15, shift back to Q0
    return (int16_t)(acc >> 15);
}

static inline uint16_t ma_update(ma_t *m, uint16_t x)
{
    m->sum -= m->buf[m->idx];
    m->buf[m->idx] = x;
    m->sum += x;
    m->idx = (m->idx + 1) & (MA_L - 1);
    return (uint16_t)(m->sum >> MA_N);
}

static inline void dac_write_10bit(uint16_t value)
{
    if (value > 1023) value = 1023;
    DAC0.DATA = (uint16_t)(value << 6);   // left adjust into 16-bit register
}

static void coeff_init(void)
{
    // Convert float coeffs to Q15
    for (uint8_t i = 0; i < N; i++) {
        b1[i] = (int16_t)(32768.0f * bf1[i]);
        b2[i] = (int16_t)(32768.0f * bf2[i]);
    }
}

/* =======================
 * ISR: sampling + demod
 * ======================= */

ISR(TCA0_OVF_vect)
{
    TCA0.SINGLE.INTFLAGS = TCA_SINGLE_OVF_bm;

    // Center around 0 (ADC midscale = 512)
    const int16_t x = (int16_t)adc_get_sample() - 512;

    buffer_push(&fir1_buffer, x);
    buffer_push(&fir2_buffer, x);

    const int16_t y1 = compute_fir(&fir1_buffer, b1);
    const int16_t y2 = compute_fir(&fir2_buffer, b2);

    const uint16_t E1 = ma_update(&ma1, abs16(y1));
    const uint16_t E2 = ma_update(&ma2, abs16(y2));

    static uint8_t sample_cnt = 0;
    sample_cnt++;

    // Sample mid in bit period: 25 ms (since fs=1kHz)
```

```
    if (sample_cnt == 25) {
        const uint8_t bit = (E1 > E2) ? 1u : 0u;

        g_bit_out   = bit;
        g_bit_ready = true;

        // Debug: show bit on DAC (0 or full scale)
        dac_write_10bit(bit ? 1023u : 0u);
    }

    if (sample_cnt >= 50) {
        sample_cnt = 0;
    }
}

/* =======================
 * MAIN
 * ======================= */

int main(void)
{
    clock_init();
    adc_init();
    dac_init();
    timer_init();
    coeff_init();

    USART3_init(115200);
    sei();

    printf("Demod starter...\r\n");

    // Byte building + sync (preamble 0x55 x3)
    uint8_t byte_acc   = 0;
    uint8_t bit_count  = 0;
    uint8_t sync_count = 0;
    bool    synced     = false;

    while (1) {
        // Atomically fetch bit_ready + bit_out
        uint8_t bit = 0;
        bool have_bit = false;

        cli();
        if (g_bit_ready) {
            g_bit_ready = false;
            bit = g_bit_out;
            have_bit = true;
        }
        sei();

        if (!have_bit) continue;

        // LSB first
        byte_acc |= (uint8_t)((bit & 1u) << bit_count);
        bit_count++;

        if (bit_count == 8) {
            if (!synced) {
                if (byte_acc == 0x55) {
                    sync_count++;
                    if (sync_count >= 3) {
```

```
                        synced = true;
                        printf("SYNC OK\r\n");
                    }
                } else {
                    sync_count = 0;
                }
            } else {
                printf("Byte: 0x%02X\r\n", byte_acc);
                if (byte_acc >= 32 && byte_acc <= 126) {
                    printf("ASCII: '%c'\r\n", byte_acc);
                }
            }

            // reset for next byte
            bit_count = 0;
            byte_acc  = 0;
        }
    }
}
```

Kodeutdrag 2: Kode for AVR128db48