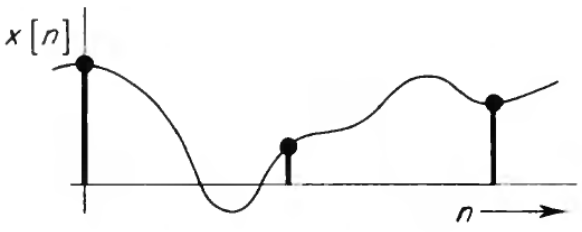
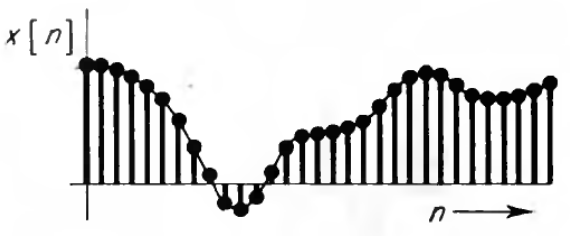
Digitale Signaal Bewerking

Practicum





HAN_UNIVERSITY OF APPLIED SCIENCES

Student : Niels Urgert (1654746)

Klas: ESE-3

Docent: ir drs E. J. Boks

Course : DSPL

Arnhem, **05-06-2024**

Inhoudsopgave

Figurenlijst

1. Inleiding

Het Digitale Signaal Bewerking (DSP) practicum had als doel het leren van de verschillende aspecten van digitaal signaalverwerking en de implementatie ervan in software. Het practicum bevat een reeks uit te voeren opdrachten, variërend van het ontwerpen van een ringbuffer tot het implementeren van filters en het genereren van frequentiebeelden.

Dit verslag biedt een overzicht van de uitvoering van het DSP practicum, inclusief de gebruikte methoden, de implementatie van de opdrachten en de behaalde resultaten.

2. Opdracht 1: Ringbuffer

In opdracht 1 van het practicum moet er een aantal klassen worden geschreven voor de implementatie van de ringbuffer. De vereiste klassen zijn een ringbufferklasse en een complex getal klasse, die afgeleid is van de imaginaire getal klasse.

Een ringbuffer is een buffer dat een bepaalde hoeveelheid data kan opslaan zonder een begin of einde te hebben. In plaats daarvan wordt de data opgeslagen in een cirkelvormige structuur, waarbij nieuwe elementen worden toegevoegd aan het einde en oudere elementen worden verwijderd aan het begin. Hierdoor kan de ringbuffer continu nieuwe data opslaan, zelfs als alle beschikbare posities al zijn gebruikt (zie Figuur 2).

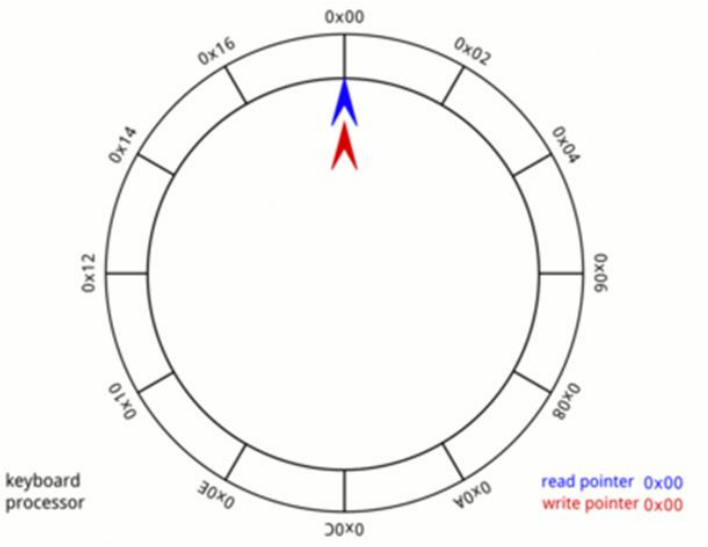
Een belangrijk kenmerk van een ringbuffer is dat het een beperkte grootte heeft. Wanneer nieuwe data wordt toegevoegd en de buffer vol is, zal het oudste element worden overschreven om ruimte te maken voor het nieuwe element. Dit betekent dat de ringbuffer altijd de meest recente data bevat binnen zijn capaciteit.

Het gebruik van een ringbuffer is vooral handig in situaties waarin een continue datastroom moet worden verwerkt en oudere data niet relevant is. Het wordt vaak gebruikt in toepassingen zoals audio- en videosignalen, waarbij real-time verwerking en een efficiënte geheugenopslag vereist zijn.

In figuur 1 wordt de ringbuffer, die in de software is geïmplementeerd, naar toegeschreven en uitgelezen.

```
waarde [0] geschreven in buffer=[Re:2,000 Im:8,000]
waarde [1] geschreven in buffer=[Re:6,000 Im:17,000]
waarde [2] geschreven in buffer=[Re:14,000 Im:30,000]
waarde [3] geschreven in buffer=[Re:26,000 Im:47,000]
waarde [4] geschreven in buffer=[Re:42,000 Im:68,000]
waarde [5] geschreven in buffer=[Re:62,000 Im:93,000]
waarde [6] geschreven in buffer=[Re:86,000 Im:122,000]
waarde [7] geschreven in buffer=[Re:114,000 Im:155,000]
waarde [8] geschreven in buffer=[Re:146,000 Im:192,000]
waarde [9] geschreven in buffer=[Re:182,000 Im:233,000]
waarde [10] geschreven in buffer=[Re:222,000 Im:278,000]
waarde [11] geschreven in buffer=[Re:266,000 Im:327,000]
waarde [12] geschreven in buffer=[Re:314,000 Im:380,000]
waarde [13] geschreven in buffer=[Re:366,000 Im:437,000]
Buffer teruglezen. De buffer is 7 elementen groot.
Buffer element [n-2] = [Re:266,000 Im:327,000]
Buffer element [n-5] = [Re:146,000 Im:192,000]
Buffer element [2] = [Re:114,000 Im:155,000]
Buffer element [5] = [Re:222,000 Im:278,000]
buffer[n-0]=[Re:366,000 Im:437,000]
buffer[n-1]=[Re:114,000 Im:155,000]
buffer[n-2]=[Re:146,000 Im:192,000]
buffer[n-3]=[Re:182,000 Im:233,000]
buffer[n-4]=[Re:222,000 Im:278,000]
buffer[n-5]=[Re:266,000 Im:327,000]
buffer[n-6]=[Re:314,000 Im:380,000]
buffer[n-7]=[Re:366,000 Im:437,000]
buffer[n-8]=[Re:114,000 Im:155,000]
buffer[n-9]=[Re:146,000 Im:192,000]
buffer[n-10]=[Re:182,000 Im:233,000]
buffer[n-11]=[Re:222,000 Im:278,000]
buffer[n-12]=[Re:266,000 Im:327,000]
buffer[n-13]=[Re:314,000 Im:380,000]
```

Figuur 2 Output Ringbuffer



Figuur 1 Ringbuffer

3. Exponentieel lopende -en lopende gemiddelde

In opdracht 2 moet de gemiddelde berekend worden van een reeks data van een ingelezen bestand. Deze data wordt berekend met twee versies van een lopend gemiddelde filter, waarbij de gebruiker het aantal punten in het filter kan instellen. Het originele signaal en het gemiddelde worden weergegeven in een grafisch venster. De twee lopende gemiddelde methoden zijn het lopende gemiddelde, zoals geïmplementeerd in opdracht 1 en een exponentieel lopend gemiddelde.

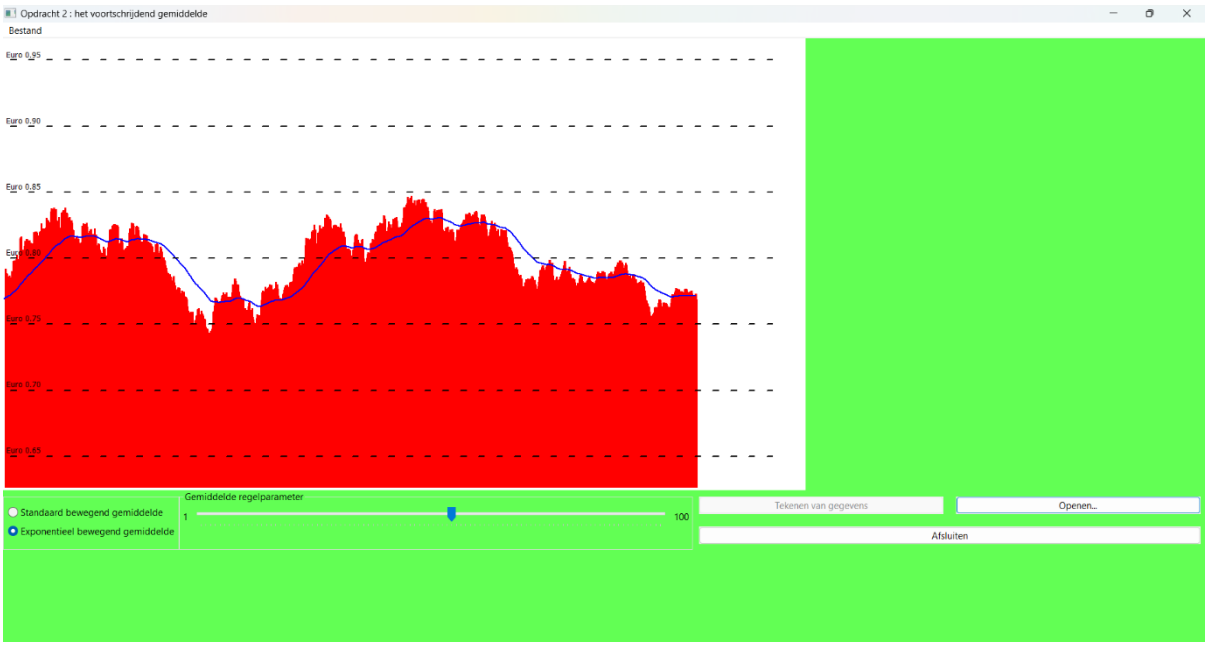
Een lopend gemiddelde is een techniek om een reeks van gemiddelde waarden te berekenen uit een reeks van data punten. Dit gemiddelde wordt steeds berekend over een vaste omvang (aantal periodes) en verschuift telkens één periode. Het doel is om ruis in de data te verminderen en de onderliggende trend beter zichtbaar te maken. De methode wordt vaak gebruikt voor het gladstrijken van data om een algemene trend te identificeren.

Een exponentieel lopend gemiddelde is een type gewogen gemiddelde waarbij de recentere waarden in de datareeks een grotere invloed hebben op het gemiddelde dan oudere waarden. Dit wordt bereikt door een exponentiële weging toe te passen, waarbij de gewichten exponentieel afnemen naarmate de data verder in het verleden ligt. De methode wordt vaak gebruikt in financiële toepassingen zoals technische analyse, omdat het beter reageert op recente prijsveranderingen. In de software worden de horizontale lijnen, y-as waardes en de gegevens uit het bestand in de grafiek getekend. Het belangrijkste is het uitrekenen van de standaard en exponentiele gemiddelde waarde. De implementatie hiervan wordt weergegeven in figuur 3.

```
auto regelparameter = avgValueSlider->GetValue();

// Regelparameter inverteren voor Exponentieel filter
int u = 0;
for (int x = 1; x <= regelparameter; ++x) {
     u = 101 - x;
}
regelparameter = u;

ExponentialAverageFilter expFilter(1.0f / (static_cast<float>(regelparameter) + 1.0f));
RingBuffer<double> ringBuffer(static_cast<unsigned short>(avgValueSlider->GetValue()));

// Geselecteerde filtertype (Standaard of Exponentieel)
if (filterType == 0) {
    // Gemiddelde signaalpunten naar variabele averageSignalPoints met behulp van ringBuffer
    for (unsigned short i = 0; i < data.GetCount(); i++) {
        const auto yAs = ((ringBuffer.gemiddelde(data.Item(i)) - 0.0f) * (1800.0f - 2.0f) / (1.0f
- 0.0f) + 2.0f);
        averageSignalPoints.Add(wxPoint(static_cast<double>(i), yAs));
    }
    grafiek->zetTekenPen(wxPen(wxColour(wxT("BLUE")), 2, wxSOLID));
    grafiek->tekenSpline(averageSignalPoints);
}
else {
    // Exponentiele signaalwaarden naar variabele expSignalPoints met behulp van expFilter
    iterator = data.begin();
    for (double xAs = 0.0f; iterator < data.end(); iterator++, xAs++) {
        const auto yAs = ((expFilter.filter(*iterator) - 0.0f) * (1800.0f - 2.0f) / (1.0f - 0.0f)
+ 2.0f);
        expSignalPoints.Add(wxPoint(xAs, yAs));
    }
    grafiek->zetTekenPen(wxPen(wxColour(wxT("BLUE")), 2, wxSOLID));
    grafiek->tekenSpline(expSignalPoints);
}
```

Figuur 3 Implementatie van de Standaard en Exponentieel gemiddelde

De werking van de software wordt in figuur 4 weergegeven. Hier in is de blauwe lijn het uitgerekende gemiddelde van de gegevens weergegeven in het rood.



Figuur 4 Output Exponentieel lopend gemiddelde

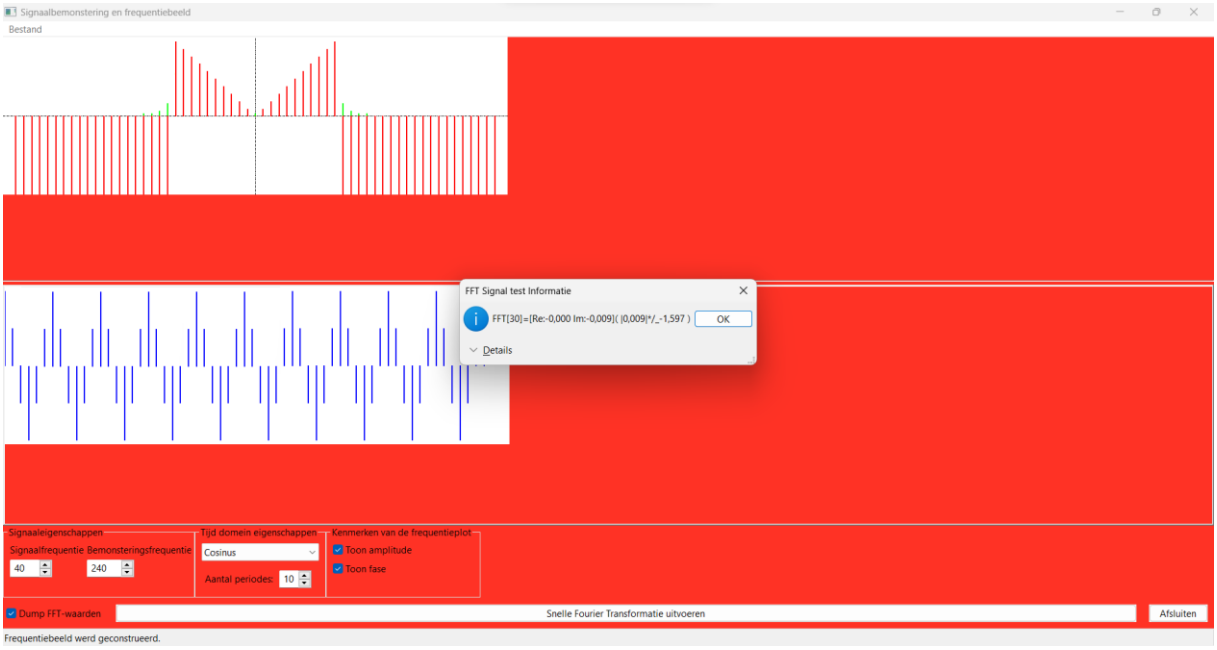
4. Opdracht 3: Fast Fourier Transform

Voor opdracht 3 is een Fast Fourier Transform (FFT) in het programma gemaakt om eenvoudig een testsignaal te kunnen instellen en dit signaal in het frequentiedomein te kunnen bekijken. Het testsignaal kan worden ingesteld op basis van de vorm (cosinus, driehoek, blokgolf of ingelezen uit een databestand), signaalfrequentie, bemonsteringsfrequentie en het aantal perioden. De code is geschreven met behulp van de FFTW toolkit om het frequentiebeeld te genereren.

De FFT is een efficiënte algoritme om een discrete Fourier-transformatie (DFT) van een reeks of een signaal te berekenen. Het vertaalt een tijd- of ruimtedomeinsignaal naar een frequentiedomeinrepresentatie. Dit betekent dat het de amplituden en fasen van de frequenties die aanwezig zijn in het signaal onthult. FFT wordt bijvoorbeeld gebruikt voor signaalverwerking om tijdsignalen in audiobewerking, beeldverwerking, en communicatie te analyseren en als filtering.

```
double* input;
fftw_complex* output;
fftw_plan p;

input = (double*)fftw_malloc(signaal.GetCount() * sizeof(double));
output = (fftw_complex*)fftw_malloc((signaal.GetCount() * sizeof(fftw_complex)) / 2 + 1);

for (auto i = 0; i < signaal.GetCount(); i++) {
	input[i] = signaal.Item(i);
}

p = fftw_plan_dft_r2c_1d(signaal.GetCount(), input, output, FFTW_PRESERVE_INPUT + FFTW_ESTIMATE);
fftw_execute(p);
```

Figuur 5 Implementatie van FFT d.m.v. FFTW toolkit



Figuur 6 Output FFT

5. Opdracht 4: Finite Impulse Response – Filter

In opdracht 4 is in het programma, de code gemaakt om een FIR-filter (Finite Impulse Response) te kunnen instellen om bepaalde signaalbanden door te laten of te blokkeren. De gebruiker kan parameters zoals de bemonsteringsfrequentie, start- en stopfrequenties van de doorlaatband, filterorde, versterkingsfactor, vensterfunctie en het aantal bits voor de fixed-point codering instellen. De applicatie moet FIR-coëfficiënten kunnen genereren op basis van de opgegeven frequentierespons, waarbij de coëfficiënten eerst in floating-point formaat worden berekend en vervolgens worden omgezet naar fixed-point (Qx-formaat). De rechthoek, driehoek en hamming functies volgens figuur 7, 8 en 10 zijn verwerkt in de software (zie Figuur 9 en 11). Hierin is n het sample en M de orde grootte.

$$w[n] = 1.$$

Figuur 7 Rechthoek venster (Dirichlet)

$$w[n] = \frac{(M+1)-|n|}{(M+1)^2}, \quad -M \le n \le M$$

Figuur 8 Driehoek venster (Barlett) [1] (5.19)

```
double FilterVenster::driehoek(const Int32 n) const
{
	double denominator = (orde + 1) * (orde + 1);
	return ((orde + 1) - fabs(n)) / denominator;
}
```

Figuur 9 Implementatie van Driehoek venster

$$w]n] = 0.54 + 0.46 \cos\left(\frac{n\pi}{M}\right), \quad -M \le n \le M$$

Figuur 10 Hamming venster [1] (5.22)

```
double FilterVenster::hamming(const Int32 n) const
{
	return 0.54 + 0.46 * cos((n * Pi) / orde);
}
```

Figuur 11 Implementatie van Hamming venster

Een FIR-filter is een type digitale filter dat wordt gebruikt in signaalverwerking om specifieke frequentiecomponenten van een signaal te bewerken of te verwijderen. Het wordt bijvoorbeeld toegepast voor audiobewerking om ruis te verwijderen, te filteren of bepaalde frequenties te versterken of verzwakken. Ook in ECG-machines wordt het gebruikt om ruis en artefacten uit hartsignalen te filteren.

Figuur 12 Output FIR-Filter

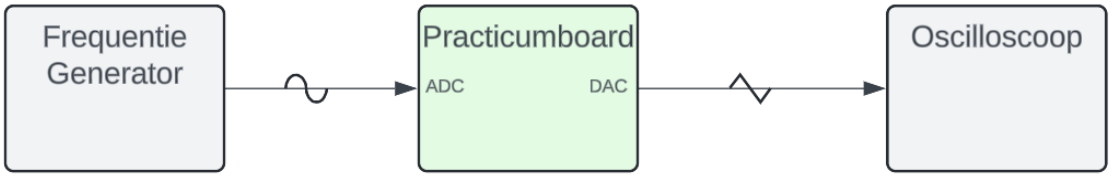
De extra optie voor het weergeven van de coördinaten in het tijdsdomein en het frequentiedomein is in de software geïmplementeerd. De implementatie hiervan wordt in figuur 13 weergegeven.

```
void FilterVenster::tijdViewMuisBewegingHandler(wxMouseEvent &event)
{
  if (true == tijdDomeinCoords->IsEnabled())
  {
    // Get the mouse coordinates
    const wxPoint mouseCoord(tijdDomeinGrafiek->converteerMuisPositie(const_cast<wxMouseEvent
&>(event)));

    // Calculate the filter effect at a certain frequency
    int n = mouseCoord.x;

    // Calculate the value of an impulse response sample
    if (n >= -orde && n <= orde)
    {
      // Calculate the value of the impulse response sample
      float sampleValue = berekenFloatingPoint(filterCoeffs[orde + n]);

      // Display the information
      wxString info = wxString::Format("Coefficient h[%d] = %.4f", n, sampleValue);
      tijdDomeinCoords->SetLabel(info);
    }
    else
    {
      tijdDomeinCoords->SetLabel("Outside range");
    }
}

void FilterVenster::freqViewMuisBewegingHandler(wxMouseEvent &event)
{
  if (true == freqDomeinCoords->IsEnabled())
  {
    // Get the mouse coordinates
    const wxPoint muiscoord(freqDomeinGrafiek->converteerMuisPositie(const_cast<wxMouseEvent
&>(event)));

    // Calculate the filter effect at a certain frequency
    double omega = muiscoord.x * (Pi / freqDomeinGrafiek->GetClientSize().GetWidth());

    double dB = muiscoord.y * (H_Omega_max - H_Omega_min) / freqDomeinGrafiek-
>GetClientSize().GetHeight();

    if(omega >= 0 && omega <= Pi)
    {
      // Calculate the index in the frequency response array
      int index = static_cast<int>((omega / Pi) * FreqSpectrumPunten(taps));

      // Ensure the index is within bounds
      if (index >= 0 && index < H_Omega.size())
      {
        // Calculate the filter effect at the given frequency
        double filterEffect = H_Omega[index];

        // Display the information
        wxString info = wxString::Format("|H(%.4f*pi)| = %.2f dB at [%.4f*pi, %.2f dB]", omega / Pi,
filterEffect, omega / Pi, dB);
        freqDomeinCoords->SetLabel(info);
      }
      else
      {
        freqDomeinCoords->SetLabel("Outside range");
      }
    }
    else
    {
        freqDomeinCoords->SetLabel("Outside range");
    }
  }
}
```

Figuur 13 Implementatie van de extra muisbeweging optie

6. Opdracht 5: Implementatie van het FIR – Filter op het practicumboard

In opdracht 5 moet de FIR-filter toepast worden op het practicumboard. De filterwaarden en het ontworpen FIR-filter uit opdracht 4 zijn nodig voor opdracht 5. Voor de opdracht hoeft alleen in de klasse genaamd STM32FilterApp, de juiste bemonsteringsfrequentie, ADC- en DAC-instellingen worden toegepast om het filter op het practicumboard te testen.



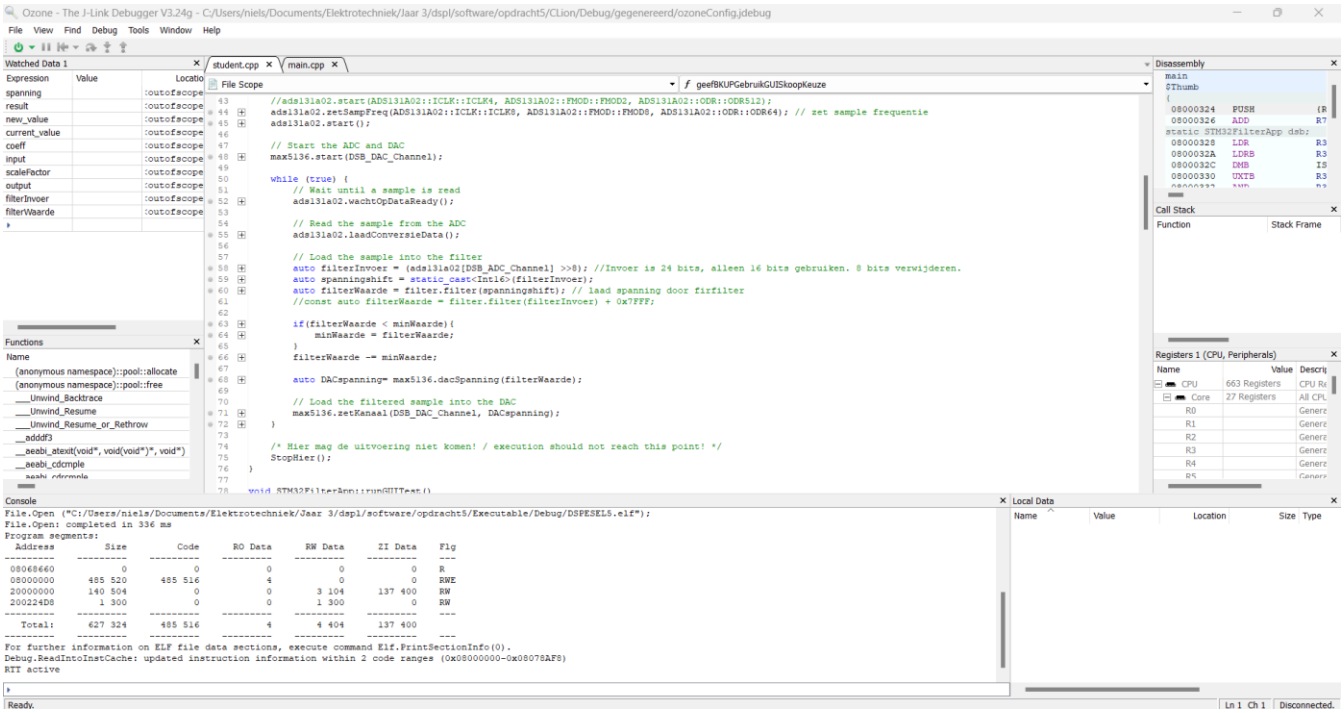
Figuur 14 Blokdiagram opstelling

De bemonstering frequentie van de ADC is ingesteld op 4 KHz. De ADC en DAC worden opgestart. De ADC wordt uitgelezen. De waarde uit de ADC wordt bewerkt door het filter, waarna de waarde uit het filter naar de DAC gestuurd.

```
void STM32FilterApp::runFilter()
{
    // Instellen van de sampling frequency van de ADC 16,38MHz / 2 / 4 / 521 = 4KHz
    ads131a02.zetSampFreq(ADS131A02::ICLK::ICLK8, ADS131A02::FMOD::FMOD8, ADS131A02::ODR::ODR64);
    ads131a02.start();

    // Start de ADC en DAC
    max5136.start(DSB_DAC_Channel);

    while (true) {
        // Wacht tot een sample gereed is
        ads131a02.wachtOpDataReady();

        // Lees de sample uit de ADC
        ads131a02.laadConversieData();

        /* Laad de sample in de filter */
        // De invoer is 24 bits. Alleen 16 bits wordt gebruikt, dus verwijder 8 bits
        auto filterInvoer = (ads131a02[DSB_ADC_Channel] >>8);
        auto spanningshift = static_cast<Int16>(filterInvoer);
        // Laad de spanning in het firfilter
        auto filterWaarde = filter.filter(spanningshift);

        if(filterWaarde < minWaarde){
            minWaarde = filterWaarde;
        }
        filterWaarde -= minWaarde;

        auto DACspanning= max5136.dacSpanning(filterWaarde);

        // Laad de gefilterde sample in de DAC
        max5136.zetKanaal(DSB_DAC_Channel, DACspanning);
    }

    /* Hier mag de uitvoering niet komen! / execution should not reach this point! */
    StopHier();
}
```

Figuur 15 Implementatie van ADC en DAC

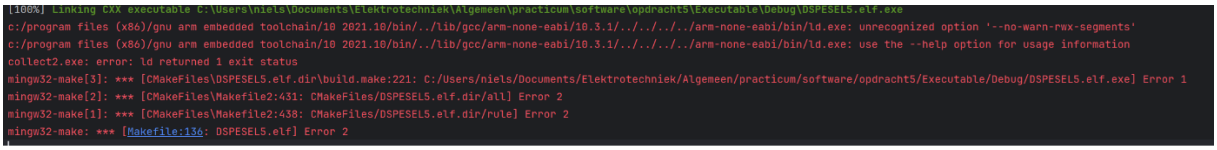
Voor de ontwikkeling van de software wordt gebruik gemaakt van de Jetbrains CLion-ontwikkelomgeving en voor het programmeren en debuggen van de embedded software op het practicumboard wordt Segger Ozone gebruikt.



Figuur 16 Ozone GUI opdracht 5

6.1 Aanpassingen

Na aanleiding van een aantal foutmeldingen in de software bestanden van DSP zijn een aantal extra aanpassingen gemaakt, anders kon opdracht 5 niet gecompileerd kon worden.



Figuur 17 Foutmelding opdracht 5

BasisSTM32.cmake

```
# Verwijdert: error unrecognized option '–no-warn-rwx-segments'
#set(StandaardLinkVlaggen "--specs=nano.specs -mthumb -Wl,--gc-sections -
Wl,-Map,${PROJEKTNAAM}.map -nostartfiles -Wl,--no-warn-rwx-segments")
```

cmsis_os.c

```
//Verwijdert: error nergens anders in de software gedefinieerd
 //xPortSysTickHandler();
```

syscalls.cpp

```
//Verwijderd: error meerdere definities van init
/*
void _init()
{

}*/
```

Figuur 18 Aanpassingen in BasisSTM32.cmake, cmsis_os.c en syscalls.cpp

7. Bibliografie

[1] P. A. Lynn en W. Fuerst, Introductory Digital Signal Processing with computer applications, John Wiley & Sons, 1989.