Journal 3

Indlejret Signal Behandling

Gruppe 1

Navn	Studienummer	Retning
Mathias Ørnstrup Hvidberg	201905706	E
Niels Højrup Pedersen	201604812	E
Jakob Saugbjerg Lange	201907544	E

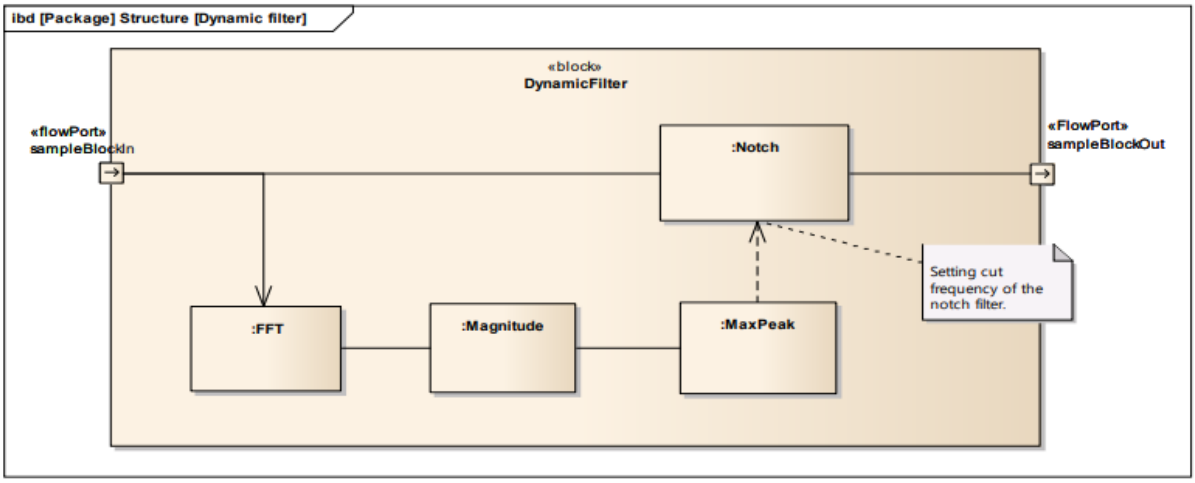
Indhold

Introduktion

I denne øvelse implementeres der et dynamisk notch-filter, hvis formål er, at fjerne den maksimale frekvens i et givent input signal. Først gennemgås SW-designet, og de vigtigste dertilhørende algoritmer, og derefter testes og dokumenteres implementeringen.

Software Design og Implementering

På det strukturelle plan, er det tiltænkt at det dynamiske notch filter skal fungere, som illustreret i IBD'et på Figur 1 nedenfor, som er lånt fra oplægget til opgaven.



Figur 1 - Blok Diagram: implementering af dynamisk Notch filter

Der tages N samples ind i blokke, som føres parallelt ud i to forskellige sektioner, hvoraf den ene er selve notch-filteret og den anden indeholder algoritmerne, der skal til for at identificere, hvilken frekvens i signalet, som har mest energi. Når en ny maxværdi er fundet, udregnes der nye koefficienter til notch-filteret, således at det kan tage hånd om denne frekvens. Denne struktur er implementeret i funktionen `DynamicFilter::Process(short * input, short* output, short len)`.

I funktionen process påtrykkes filteret, hvorefter der udføres en FFT på inputtet ved brug af et twiddle table. Twiddle table indeholder alle de elementer der bruges i fourier-transformationen, som er konstante, hvorved FFT ikke skal andet et at slå dem op i tabellen. Til sidst kaldes findMax-funktionen, som finder frekvensen for den maksimale magnitude af FFT'en. Herefter sættes notch-filterets cutfrekvens til denne frekvens for maksimum, hvormed de nye filter-koefficienter kan udregnes. Maksimummet skal dog være over et givent threshhold, der gives som parameter til findMax-funktionen. Koden fremgår nedenfor af Listing 1.

```
void DynamicFilter::process(short* input, short* output, short len)
{
	int block_exponent;

	// Perform notch filtering
	m_IIRFilter.process(input, output, len);

	// FFT
	rfft_fr16(input, m_fft_output, m_twiddle_table, 1,
					N_FFT, &block_exponent, 2);

	// Compute magnitude
```

```
        fft_magnitude_fr16(m_fft_output, m_real_magnitude,
                                        N_FFT,  block_exponent, 1);

        findMax(PEAK_THRESHOLD);
 }
```

Listing 1 - DynamicFilter::Process funktion

Blokken ":MaxPeak" er implementeret som set nedenfor. Den fungerer således at, alle magnituder fra FFT-udregningen gennemløbes, og hver gang der findes en ny "størst" værdi, assignes denne til en variabel `fract16 max`. Når hele arrayet er blevet gennemløbet, tjekkes det hvorvidt den nye max værdi, er over et givent threshold, som her er defineret til værdien 3000. Hvis dette er tilfældet, udregnes den pågældene frekvens (fnotch), ud fra bin-nummeret og frekvensopløsningen:

$$f_{resolution} = \frac{fs}{N} = \frac{48kHz}{1024} = 46.875\ Hz$$

Eksempel på fnotch udregning:

$$f_{notch} = f_{resolution} \cdot N_{bin} = 46.875 \cdot 20 = 937.5\ Hz$$

Hvis denne frekvens ikke er lig med den sidst fundne max-frekvens, sættes flaget `m_updateNotch`, som medfører at nye koefficienter beregnes.

```
#define PEAK_THRESHOLD   3000

// Find maximum peak in FFT magnitude response
void DynamicFilter::findMax(fract16 threshold)
{
        short i, i_max;
        fract16 max = 0;

        // TODO Verify and improve code below to
        // find maximum amplitude in frequency spectrum
        for (i = 1; i < FFT_SIZE; i++)
        {
                if (m_real_magnitude[i] > max)
                {
                        i_max = i;
                        max = m_real_magnitude[i_max];
                }
        }

        // Check maximum peak above threshold
        if (max >= threshold)
        {
                float fnotch = i_max * ((float)m_sampleRate / N_FFT);
                if (fnotch != m_fnotch)
                {
                        m_fnotch = fnotch;
                        // Signal to main loop update notch filter
                        m_updateNotch = true;

                        //printf("New max found: %f Hz\n", fnotch);
                }
        }
}
```

Listing 2 - Implementering af FindMax, som identificerer frekvens med mest energi

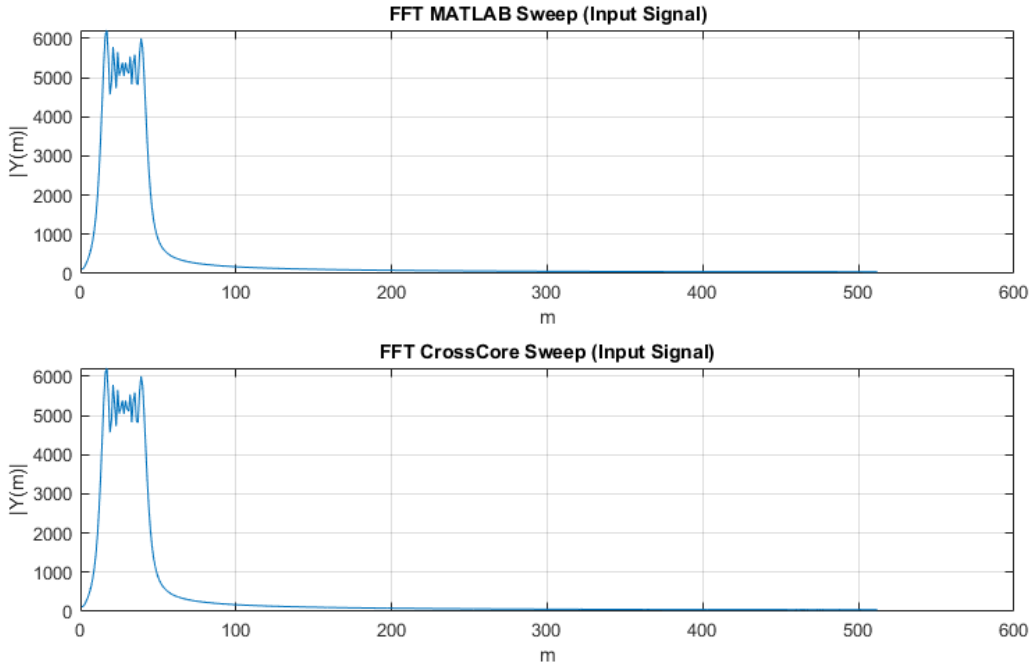
Tjek af flaget sker i `DynamicFilter::updateDynFilter(void)`, som kontinuerligt kaldes fra main-loopet. Hvis det er sat, kaldes funktionen makeNotch i IIRFilter-objektet, som medfører nye koefficienter.

```
void DynamicFilter::updateDynFilter(void)
{
	if (m_updateNotch)
	{
	// TODO Change code to handle update of notch filter when new peak found
		m_IIRFilter.makeNotch(m_sampleRate, m_fnotch, 0.95);

	//m_IIRFilter.makeNotch(m_sampleRate, 1000, 0.95); //
		m_updateNotch = false;
	}
}
```

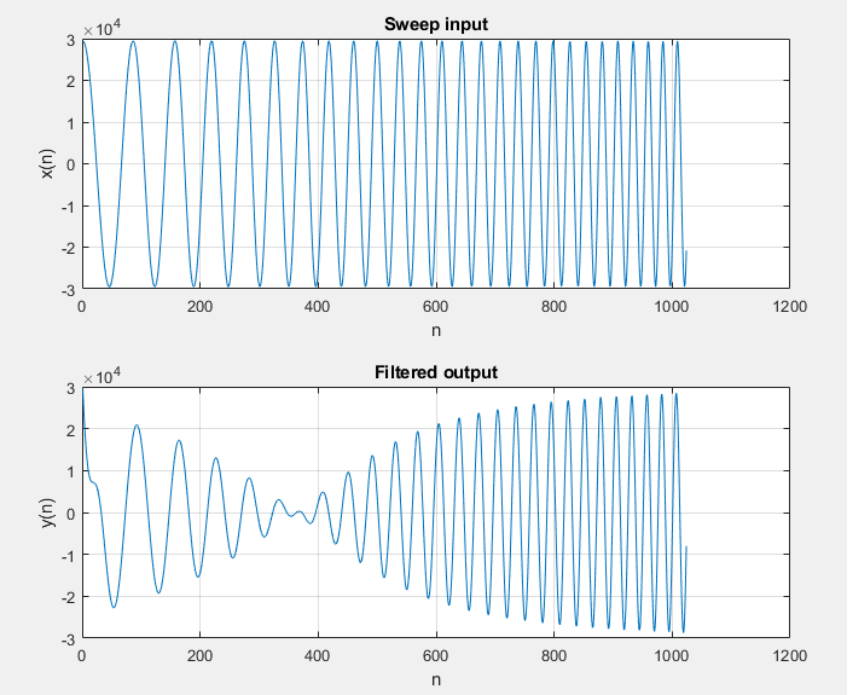
Verificering af indlejret FFT og notch-filteret

I matlab laves et sweep, se Figur 3, til at teste programmets FFT. Øverst på Figur 2 ses FFT udført i matlab med den indbyggede funktion fft(). Der benyttes samme samplingsfrekvens og antal samples. Det nederste billede i figuren viser FFT'en udført af implementeringen på blackfin. De to amplitude-responser er plottet i matlab og er identiske, hvormed det kan konkluderes at den implementerede FFT er funktionsdygtig.



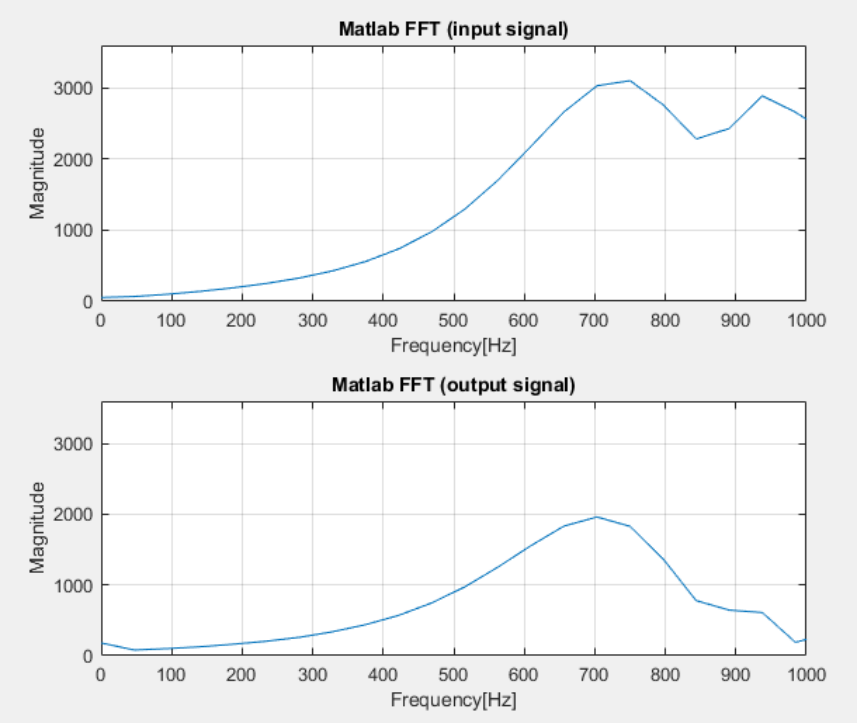
Figur 2 - FFT af sweep i Matlab og i DSP-implementeringen

På Figur 3 ses sweep signalet genereret i matlab som føres ind i blackfin, samt det filtrerede signal outputtet fra blackfin plottet i matlab. Filteret er i denne test ikke dynamisk, men sat til en konstant cutfrekvens på 1000Hz, for at teste at filteret virker.



Figur 3 - Sweep inputtet genereret i matlab og det dertilhørende filtrerede signal udlæst fra blackfin. Filteret har i denne simple test en cutfrekvens på 1000Hz

På Figur 4 ses FFT af de to signaler. På Figur 3, ses det at signalet ved 1000Hz dæmpes som forventet af notch-filteret, hvormed det kan konkluderes at notch-filteret er korrekt implementeret.



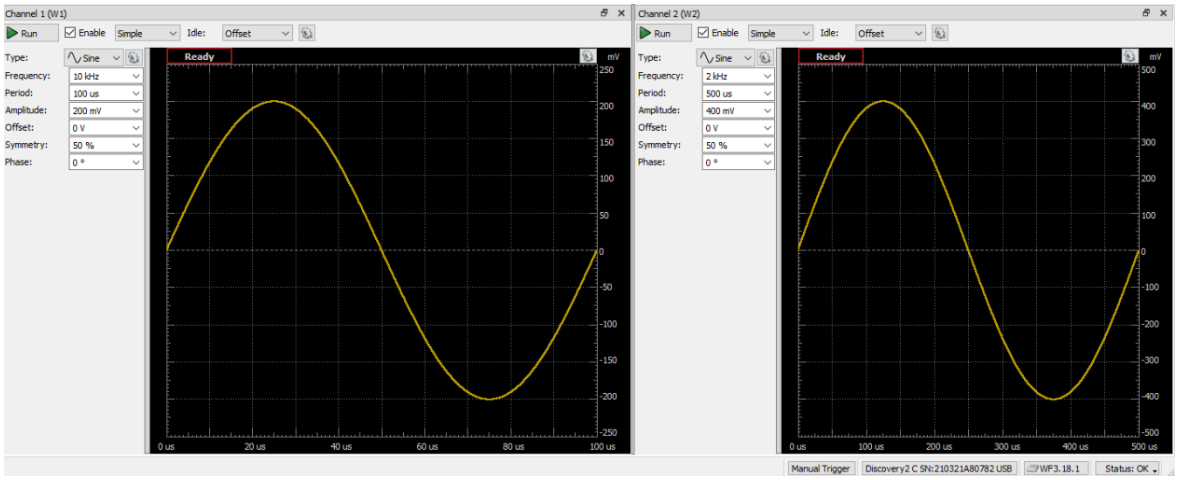
Figur 4 - FFT af input-sweep-signalet og af output-signalet

I realtime scenariet, ville denne frekvens blive detekteret som en 984.375Hz tone, pga. af frekvensopløsningen, hvorfor resultatet her, sandsynligvis ikke vil blive det samme, da

koefficienterne i så fald ville give et lidt andet notch-filter en ønsket. Derfor ville det også give mening ikke at sætte pol-radius for høj, så at filteret ikke bliver alt for skarpt.

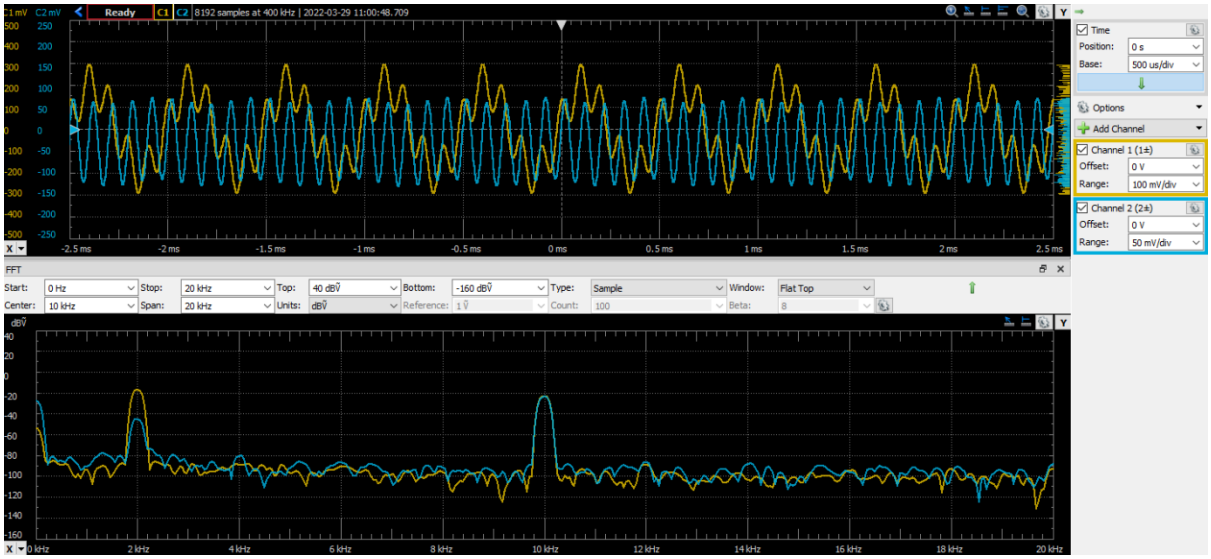
Verificering af dynamisk filterfunktionalitet

Ved at føre et testsignal igennem filteret, kan det eftervises at filteret dæmper den del af testsignalet, med højeste magnitude. På Figur 5 ses et tilfældigt testsignal, hvor to sinussignaler bliver summeret sammen med to forskellige amplituder og to forskellige frekvenser.



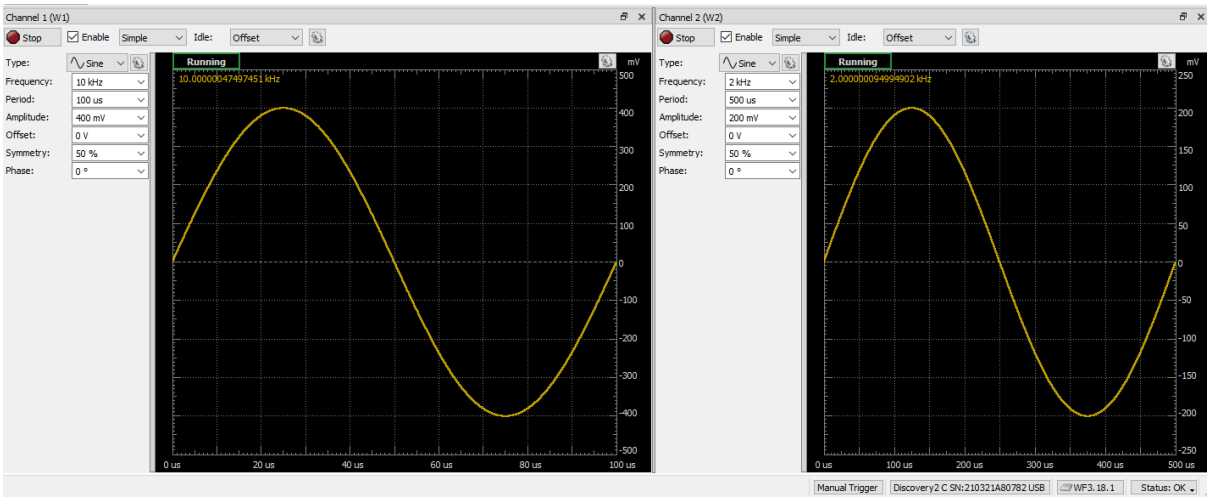
Figur 5 - To sinus signaler med en amplitude på henholdsvis 200mV og 400mV, samt tilhørende frekvenser på 10 kHz og 2KHz

På Figur 6 ses resultatet af filteret. Øverst ses resultatet i tidsdomænet, hvor det gule er summen af de to sinussignaler, og det blå er outputtet fra filteret. Nederst ses resultatet af filteret i tidsdomænet. Her fremgår det at de 2 kHz bliver dæmpet, hvor de 10 kHz kommer upåvirket igennem. Dette giver også mening, da testsignaler med 2 kHz har størst amplitude.



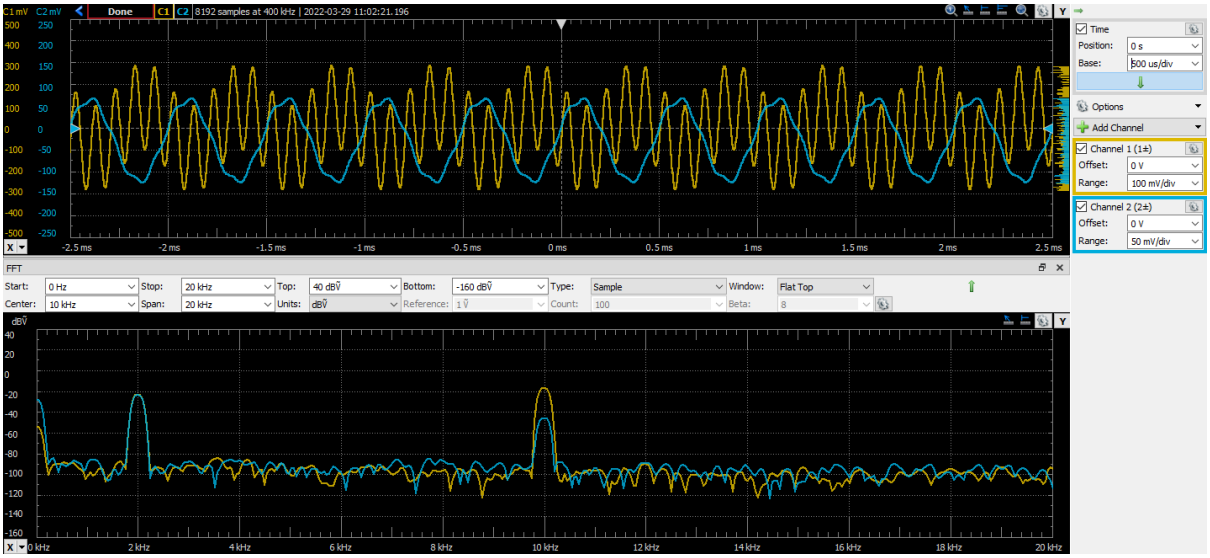
Figur 6 - Frekvensrespons for testsignalet, hvor 2kHz signaler bliver dæmpet. Det blå signal er output og det gule signal er input.

På Figur 7 ses det samme testsignal som på Figur 5, dog er der byttet om på amplituderne.



Figur 7 - Samme testsignal, dog er amplituden på de to signaler bygget om, i forhold til Figur 5.

På Figur 8 ses resultatet af filteret dom dæmper de 10 kHz fremfor de 2 kHz. Det giver også mening, da magnituden af signalet ved de 10 kHz er størst.



Figur 8 - Frekvensrespons for testsignalet, hvor 10 kHz signaler bliver dæmpet

I Listing 3 ses en opdateret funktion: `DynamicFilter::process`, hvor den originale er præsenteret i Listing 1. Ved at udnytte `<cycle_count.h>` kan antallet af clock-cycles udskrives. På den måde kan det estimeres hvor lang tid det tager at udføre en funktion. Optællingen af cycles udføres for hver hundrede eksekvering af funktionen, for ikke at udskrive og tælle ved hvert interrupt.

```
uint16_t counter = 0;

void DynamicFilter::process(short* input, short* output, short len)
{
	int block_exponent;

	if(counter == 100)
		CYCLES_START(m_stats);

	// Perform notch filtering
	m_IIRFilter.process(input, output, len);

	// TODO add code to perform FFT and magnitude and call findMax
	rfft_fr16(input, m_fft_output, m_twiddle_table, 1, N_FFT, &block_expo-
nent, 2);

	fft_magnitude_fr16(m_fft_output, m_real_magnitude, N_FFT, block_exponent,
1);

	findMax(PEAK_THRESHOLD);

	if(counter == 100){
		CYCLES_STOP(m_stats);
		CYCLES_PRINT(m_stats);
		CYCLES_RESET(m_stats);
		counter = 0;
	}
	counter++;

 }
```

Listing 3 - DynamicFilter::Process funktion med Cycles counter enabled.

På Figur 9 ses resultatet af koden i Listing 3.

```
Application running: SW4 turns filter on, SW5 turns filter off
        CYCLES : 325031
        CYCLES : 323255
        CYCLES : 318554
        CYCLES : 323313
        CYCLES : 322897
        CYCLES : 318523
        CYCLES : 319833
```

Figur 9 - Antallet af cycles det tager at udføre funktionen process i DynamicFilter