Datorteknik
Exempel på stegvis förfining
—
Signalbehandling med AVR

Michael Josefsson

Version 0.11 2018-05-28

Innehåll

1 Inledning **3**

2 Digitalt filter på ATmega328 **4**

3 Koppla, löd och provkör delarna **5**

4 Få ihop något som funkar **6**

5 Förfina **9**

6 Bygg på funktionalitet! **14**

1 Inledning

Denna text beskriver (på ett ganska löst och översiktligt sätt medges) och diskuterar exempel på en inkrementell konstruktionsstrategi. I detta fall var det osäkert om, eller hur bra, det hela skulle fungera till slut så det var inte möjligt att skriva en initial kravspecifikation. Texten tar fasta på att man lär sig mer om ett problem efter att ha stångats mot det ett tag.

Erfarenheten om hårdvaran och problemet bakas sedan under utvecklingen in i konstruktionen:[1]

Figur 1.1: All ingenjörsmässig problemlösning handlar om stegvis förfining från en given, än så länge bristfällig, funktion.

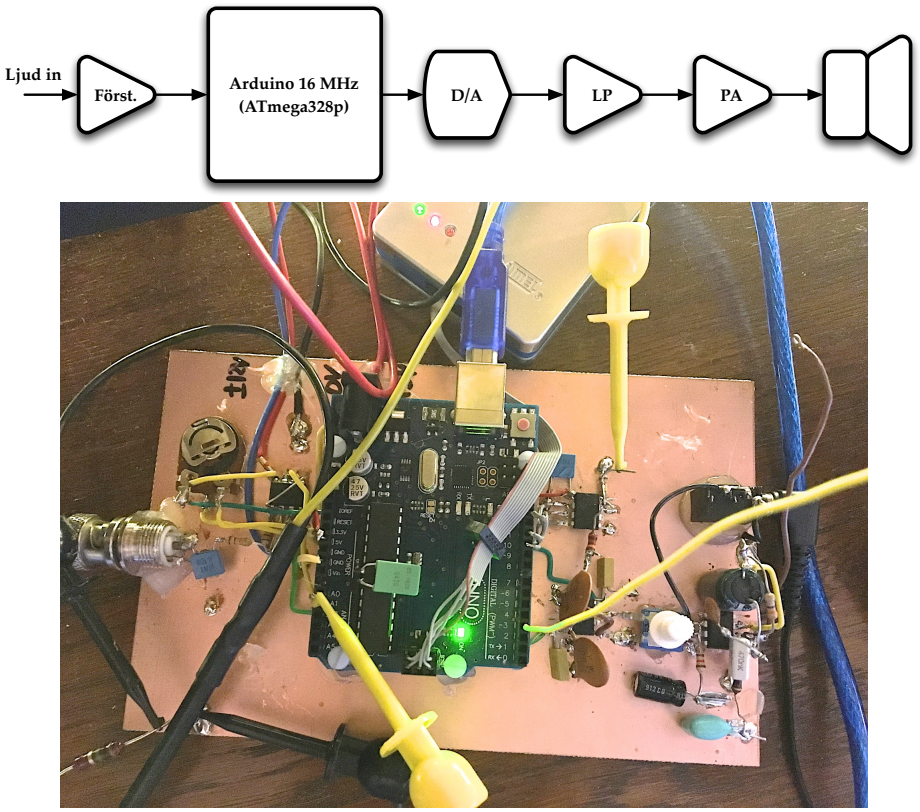
Texten är skriven i möjligen lite slapp, icke-formell stil. Det är avsiktligt och min förhoppning är att det ger lite inblick i att allt inte är klart genomtänkt från början. Det fanns en idé som jag ville testa, men i övrigt gick det mesta på fritid och ganska lösa boliner.

[1] Figuren från `https://hackaday.com/2017/11/30/life-on-contract-how-much-do-i-charge/`

2 Digitalt filter på ATmega328

För att undersöka om det går att utföra digital signalbehandling på en 8-bitars AVR-processor vid blott 16 MHz lödde jag ihop ett Arduino-kort med lite enkel kringelektronik.

Det handlade om ett ingångssteg med lite förstärkning, en snabbare digitalomvandlare än vad ATmega328p har internt, ett efterföljande lågpassfilter och en audioförstärkare för att kunna koppla in hörlurar eller högtalare till utsignalen, se figuren nedan.



Figur 2.1: Överst visas signalkedjan från insignal till högtalare. Signalen omvandlas till digitala värden i den interna A/D-omvandlaren i ATmega328p, behandlas i programvara innan de digitala värdena påförs D/A-omvandlare (D/A), lågpassfiltreras (LP) och slutligen förstärks (PA). Underst visas realiseringen i form av lödda komponenter på kopparlaminat. Utsignalen är kraftig nog att höras (ljudligt!) i en ansluten högtalare.

3 Koppla, löd och provkör delarna

Det exakta elektriska schemat är ointressant för oss här. Det viktiga är att beskriva hur uppkopplingen av hårdvara skedde på ett strukturerat sätt, steg för steg med täta prov av funktionerna:

- I detta fall lödde jag ihop ingångssteget först (längst till höger i figur 2.1) och kontrollerade att att förstärkningen och DC-offset kunde ändras med två potentiometrar som avsett. Förstärkningspotentiometern ersattes slutligen med ett fast motstånd.
- Sedan monterades Arduinokortet med smältlim och D/A-omvandlaren löddes på plats. Den D/A-omvandlare som används styrs med serieprotokollet SPI så testkod för att skicka ut något på SPI skrevs och att *något* hände på avsedda utgångar mättes med oscilloskop. Exakt *vad* som hände brydde jag mig inte om utan räknade kallt med att om rätt saker kom till D/A-omvandlaren skulle något hända på *dess* utgång.[1]
- Efter detta kopplades SPI till D/A-omvandlaren. Men en oscilloskopprob på omvandlarens utsignal och en programloop som skickade ut en uppräknad byte från `$00` till `$FF` förväntade jag mig en spänningsramp på oscilloskopet. Efter lite databladsläsning både för processorn och MPC4802 blev det just precis så.
- Sedan kopplades de rent analoga resterande delarna upp också. Med signalgenerator påfördes efter hand sinusvågor av olika frekvens för att konstatera att LP-filtret fungerade som avsett med en brytfrekvens i storleken några kilohertz och att ljudet hördes i högtalaren. Mot det ritade schemat var enda ändringen att signalen behövde dämpas ordentligt innan volymkontrollen. Lite experimenterande gav att ett 68k motstånd i serie med LP-filtrets utgång blev bra. Lägg märke till detta tillvägagångssätt: Koppla lite, prova att det blev som tänkt, åtgärda eventuellt, koppla lite, prova att det blev som tänkt, åtgärda eventuellt, koppla lite, prova att det blev som tänkt, åtgärda eventuellt, koppla lite, prova att det blev som tänkt, åtgärda eventuellt, koppla lite, ... Hela tiden små steg.

Vid det här laget visste jag att hela kedjan av hårdvara fungerade som avsett!

[1] Detta är ett rätt stort steg och motiveras av erfarenhet. Hade jag var mer noggrann hade jag försökt identifiera att det som hände på SPI-utgångarna verkligen var det som programmet skickade. För att kunna se det på ett oscilloskop skulle programmet behöva skicka exakt samma byte om och om igen. Ingen orimligt steg, men jag hoppade över det den här gången. Väl medveten om att faktiskt behöva göra det (suck!) om det strulade senare.

4 Få ihop något som funkar

Innan några egentliga signalbehandlingsrutiner kunde skrivas behövs ett antal små rutiner för att hantera hårdvaran. Det handlar om att konfigurera in- och utgångar, SPI, AD-omvandlare och interna timers.

D/A-omvandlarrutin D/A-omvandlaren är en MCP4802[1] vilken kräver SPI-kommunikation. Lyckligtvis innehåller ATmega328p hårdvara för SPI så för denna del behövs några I/O-register konfigureras. Som nämnts måste MCP4802:s datablad konsulteras för att reda ut exakt *vad* som skall skickas till den och ett program måste skrivas för att testa att den är korrekt inkopplad och att vi kan styra den. Det första programmets syfte var att *enbart* resultera i en subrutin för att skriva till omvandlaren. Efter lite hyfsning nöjde jag mig med nedanstående kod som en fristående subrutin:

```
        ; --- DAC MCP48x2 send r16 (0..FF)
        ;
dac:
        push    r17
        swap    r16
        mov     r17,r16
        andi    r17,$0F
        ori     r17,$10
        cbi     PORTB,2         ; SS on device low
        out     SPDR,r17        ; send high byte
dac1:   in      r17,SPSR        ; get status
        sbrs    r17,SPIF        ; wait
        jmp     dac1
        out     SPDR,r16        ; send low byte
dac2:   in      r17,SPSR        ; get status
        sbrs    r17,SPIF        ; wait
        jmp     dac2
        sbi     PORTB,2         ; SS on device high
        pop     r17
        ret
```

Den uppmärksamma läsaren noterar att några rader återkommer här. Så här tidigt i utvecklingen är jag dock nöjd om det fungerar även om det inte är optimalt. I detta sig-

[1] https://www.electrokit.com/mcp4802e-p.51070

nalbehandlingsfall är det väsentligt med snabb kod så jag är inte benägen att onödigtvis slöa ner rutinens körtid.

Avbrott Hela samplingsmekanismen är hårdvarustyrd. En intern timer (`tmr0`) användes för att ge periodiska avbrott med en frekvens av 62.5^2 kHz / 8 = 7.812 kHz och starta en AD-omvandling. När omvandlingen är klar (efter 13 klockpulser av AD-klockans frekvens) triggas ett `adcConversionComplete`-avbrott. Det är i denna avbrottsrutin som det samplade värdet hämtas, eventuellt behandlas och skickas ut till D/A-omvandlaren.

För att, enligt rubriken, först få något som funkar ihop, utfördes denna avbrottsrutin på enklaste sätt, det vill sägaden enbart reläade inlästa samplade värden direkt till DA-omvandlaren:

```
        ; --- ADC conversion complete
        ;
adcConversionComplete:
        push    r16
        lds     r16,SREG
        push    r16

        lds     r16,ADCH        ; r16 = 0..256
        call    dac             ; output channel

        pop     r16
        sts     SREG,r16
        pop     r16
        reti
```

Notera: Processorn ATmega328p har en del av sina I/O-register på ställen som inte kan nås med `in`- och `out`-instruktionerna. Därför används `lds` och `sts` som ersättning i denna processor. ATmega16 är väluppfostrad i detta sammanhang så där kan man använda de vanliga I/O-instruktionerna.

Nu hade jag något som samplade ljudet från min iPhone och skickade ut det. Hädanefter vill jag enbart tillföra förfiningar som inte förstör, efter varje liten ändring laddade jag ner koden till processorn för att konstatera att inget var trasigt.

[2] 16 Mhz / 256 = 62.5 kHz

En första modifiering var att markera för mig själv var signalbehandlingskoden skulle ligga. Samtidigt ville jag inte frånhända mig möjligheten att återigen kontrollera om hårdvaran fungerade som avsett, så labeln `passThru:` lades till:

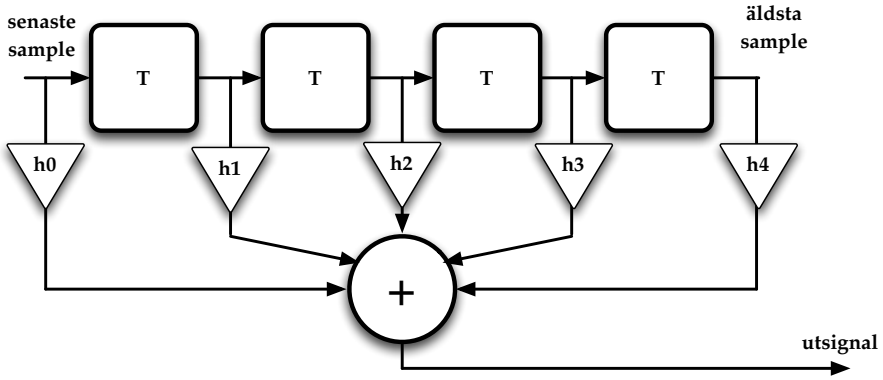
```
        ; --- ADC conversion complete
        ;
adcConversionComplete:
        push    r16
        lds     r16,SREG
        push    r16

        lds     r16,ADCH        ; r16 = 0..256
        jmp     passThru
        ;
        ; space for processing code
        ;
passThru:
        call    dac             ; output channel

        pop     r16
        sts     SREG,r16
        pop     r16
        reti
```

Hädanefter kan jag välja att köra signalbehandlingsdelen eller inte genom att kommentera bort raden `jmp passThru`. Det är överraskande praktiskt att ha sådana kortslutningar av i koden för att snabbt kunna konstatera om hårdvaran fortfarande fungerar. Här var det oftast för att konstatera att sladden mellan ingångssteget och arduinokortet lossnat igen...

5 Förfina

Digital signalbehandling är inte speciellt förlåtande: Man **HÖR** när något gått snett! Tro mig... Skickar man ut fel byte av misstag, eller om beräkningarna tar för lång tid och avbrottsrutinen avbryter vid fel tid och så vidare, hörs det omedelbart. Så resten av programutveckligen skedde nästan hela tiden med musik i högtalaren som tecken på att senaste versionen av programmet fungerar. Vilket är trevligt på mer än ett sätt.

FIR-filter En lämplig förstaövning för att bedöma processorns lämplighet i signalbehandlingssammanhang är att implementera ett FIR-filter. Ett FIR är i princip en medelvärdesbildare av flera samplade invärden. Ett "2-tap" FIR-filter leverar för varje nytt inkommet sample medelvärdet av detta och det förra samplet. Här implementerades först ett 5-tap-filter:



Figur 5.1: Filtrets utsignal beräknas för varje inkommit nytt sample. Om konstanterna h_i är lika med 1 genom antalet tap (här $\frac{1}{5}$) utför FIR-filtret en ren medelvärdesbildning, en sorts lågpassfiltrering. Med fouriertransformering kan koefficienterna beräknas för andra filterfunktioner. Med snabba signalprocessorer är flera hundra tappar möjliga.

Samtliga samplevärden är 8-bitars tvåkomplementskodade tal och ligger i en lista i SRAM mellan en startadress `$0200` och slutadress `$0200 + Buff_Size`.

Koden för filtret traverserar listan, multiplicerar de lagrade samplevärdena med $h_0, h_1, \ldots$ enligt figuren samt summerar dem i `r25:r24`.

Bra program använder pekare. Det ger tidigt en stor flexibilitet och är används pekare från början: `XH:HL` pekar ut rätt sample i bufferten och `ZH:ZL` pekar ut korrekt multiplikator ur tabellen `fir5Tab`. Avslutningsvis lagras höga delen av resultatet i SRAM i positionen `firOut`.

```
        ;
        ; --- fir5 5-tap FIR-filter
        ;
        ; lastSample(x4)*h0+butlast(x3)*h1+..+oldest(x0)*h4
        ; build in r25:24.
        ; store result in firOut (8bit)
fir5:
        push    r18
        push    XL
        clr     r24             ; sum = 0
        clr     r25
        ldi     ZH,HIGH(fir5Tab*2)
        ldi     ZL,LOW(fir5Tab*2)
        ldi     XL,Buff_Size    ; adress of last (one beyond lastSample)
fir51:
        ld      r16,X           ; lastSample
        lpm     r18,Z+
        muls    r16,r18         ; r1:r0
        add     r24,r0
        adc     r25,r1
        dec     XL
        brpl    fir51
        ;lsl    r25
        sts     firOut,r25
        pop     XL
        pop     r18
        ret
fir5Tab:
        .db     31,31,31,31,31
```

Med denna kod genomfördes ett antal tester för att studera filtrets effektivitet men framför allt för att utröna om det är tillräckligt med 8-bitars värden både som samples och konstanter. Och detta hela tiden med hänsyn till tidsåtgång.

Prov med 10-bitars sample gav att de extra bitarna inte verkade spela någon roll — antingen försvann de i beräkningarna pga avrundningsfel eller så påverkade de ändå inte den 8-bitars D/A-omvandlingen — så det bestämdes att hålla kvar 8-bitars sample.

Däremot verkade det (och inte helt entydigt här) som att 16-bitars värden var av godo för filterkoefficienterna. Att använda både 10-bitars sample *och* 16-bitars koefficienter bedömdes ta för lång beräkningstid.

En modifierad kod för de beslutade 8-bitars sample och 16-bitars konstanter blev till slut:

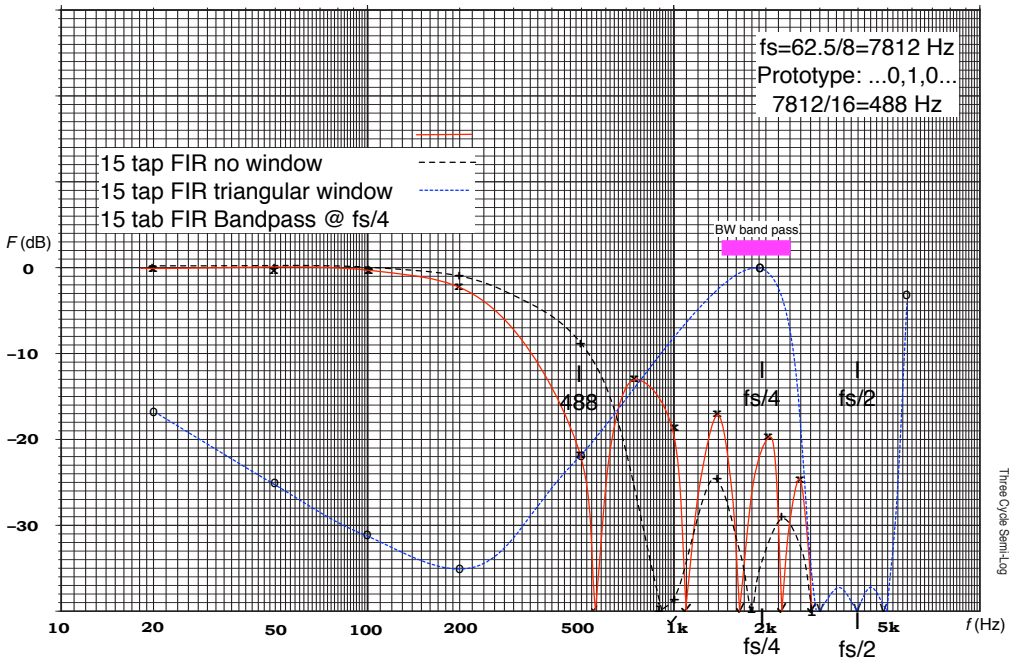
```
fir5:
        push    r18
        push    XL

        clr     r22             ; sum = 0
        clr     r23
        clr     r24
        ldi     ZH,HIGH(firTab*2)
        ldi     ZL,LOW(firTab*2)
        ldi     XL,Buff_Size    ; adress of last (one beyond lastSample)
fir51:
        ld      r16,X           ; lastSample 2's compl
        lpm     r18,Z+          ; r19:r18 16bit 2's compl constant
        lpm     r19,Z+
        lsl     r18             ; extra gain
        mulsu   r16,r18         ; Multiply signed with signed
        add     r22,r0
        adc     r23,r1
        adc     r24,zero
        mul     r16,r19         ; Multiply unsigned
        add     r23,r0          ; Add without carry
        adc     r24,zero        ; collect carry
        dec     XL
        brpl    fir1
        ;lsl    r25
        sts     firOut,r23

        pop     XL
        pop     r18
        ret
fir5Tab:
        .dw     125,125,125,125,125,125
```

Där additionen sker löpande i ett 24-bits register som utgörs av `r24:r23:r22`. Jag är osäker på om `r24` någonsin används, det handlar om storleken på `fir5Tab`-värdena, men får stå kvar tills vidare.

Bli bekant med problemet Med koden i detta skick utfördes sedan åtskilliga prov med upp till 31 tappars filter enbart för att jag skulle bekanta mig med koden och systemet. Det var lätt att höra vad som verkligen spelade roll och vad som var mindre viktigt.

För att inte få för ”kornigt” ljud i högtalaren var det till exempel viktigt att ha så stora variabelvärden som möjligt i alla beräkningar, dock utan att få overflow vid något enstaka

tillfälle, för då knäpper det obehagligt starkt i högtalaren. Med för små konstanter å andra sidan blir de (konstanterna) alla ungefär lika stora vilket gör dels att alla filter liknar varann och dels att blir utsignalen svag (ibland så svag att den blir 0, det vill säga tysta partier försvinner helt!).



Figur 5.2: Uppmätt frekvensgång hos 15-taps FIR-filter med utsignalen normerad till 0 dB. Filtret är designat efter ett prototypfilter med så smal bandbredd som möjligt (488 Hz). De faktiska filterkoefficienterna beräknades genom FFT och skalades så att overflow inte uppstod i beräkningarna. Den röda kurvan visar frekvensgången med omodifierade konstanter, den streckade med konstanter viktade med ett triangelfönster. Den blå kurvan är resultatet efter att konstanterna multiplicerats med $+1, 0, -1, 0, +1, \ldots$ vilket enligt teorin skall förflytta hela filtret till samplingsfrekvensen/4, $f_s/4$

Rensa kod och fastställ omgivningarna Under hela denna viktiga experimentperiod ändrades kommentarerna i koden allteftersom så att det alltid betydde rätt sak. Onödiga `push`/`pop` togs bort. Överhuvudtaget blev jag efter hand mer och mer bekant med koden och teorin bakom signalbehandlingen. Det känns skönt att teorin bekräftas av experimenten (med undantag för bandpassfiltret ovan. Jag undrar fortfarande om det blev helt rätt?).

Det kändes efter ett tag jobbigt att ha alla varianter av testade filter: `fir5`, `fir7`, `fir15` och `fir31` i koden varför en enda `fir`-rutin implementerades.

Initialt hade jag felaktigt förutsett att jag skulle behöva nollställa samplebufferten, plocka ut enstaka värden ur densamma och utföra en 16x16→32 bitars multiplikation. De rutiner jag skrivit för detta rensades ut. Annan död kod togs bort och det hela snyggades till så att det blev mer läsbart.

Då jag från början var osäker på vilken samplingstakt som var möjlig (egentligen tidsåtgången för beräkningar mellan samplen) provades samplingstakterna $f_s = 62.5$ kHz och $f_s = 62.5/8 = 7.812$ kHz. Den senare valdes.

Experiment undertogs också med olika hastigheter på ADC-omvandlingens klockfrekvens. Databladet anger 200 kHz som högsta frekvens men öppnar även lite diffust för högre frekvenser om försämrad precision kan tillåtas. Det visade sig att 1 MHz fungerar utan hörbar försämring av ljudet.

I och med dessa avslutande designbeslut för man betrakta denna första ”proof of concept”-fas som över. Nu finns en grund att bygga vidare på där man vet att

- den analoga hårdvaran fungerar, kanske inte ner till LSB i D/A-omvandlingen, men tillräckligt bra för att fortsätta utveckla på. Ett korrekt designat mönsterkort ger säkert bättre signaler ifråga om brum och brus och därmed ljudkvalitet. Det blir i vart fall inte sämre.
- mikrokontrollerns AD-omvandlare och interna timrar fungerar på ett tillfredsställande sätt, SPI likaså.
- med en samplingsperiod på $1/7812 = 128\,\mu s$ hinns en hel del beräkningar med. Olika uppmätta ungefärliga tider under projektets gång äro:

Åtgärd	Tidsåtgång μs
`dac`	5
A/D-omvandling	13
FIR 5 tap	9
FIR 7 tap	12
FIR 15 tap	22
FIR 31 tap	42

  Macrot

```
.macro bit3
        sbi     PORTD,3
        cbi     PORTD,3
.endmacro
```

  användes flitigt som skvallersignal för att bestämma dessa tider.

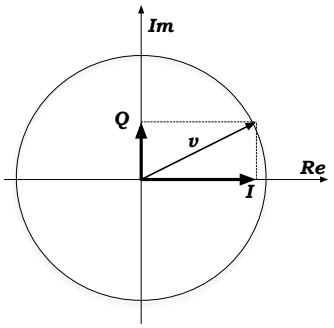
Hädanefter handlar allt om att skriva kod!

6 Bygg på funktionalitet!

Bortsett från att implementera filter enligt ovan kan man utföra mycket intressant signalbehandling om insignalen kan delas upp två ortogonala delar, en *in-phase*- och en *out-of-phase*-signal.

Den senare kallas ofta *quadrature-phase* varför de brukar förkortas till signalerna I och Q. De kan då ses som vektorer $\vec{v} = \mathsf{I} + j\mathsf{Q}$ i det komplexa talplanet vilket avsevärt underlättar till exempel studier av signalens faslägen vilket i sig öppnar för avkodning av fasmodulerade signaler.

Längden av $\vec{v}$, $|v| = \sqrt{\mathsf{I}^2 + \mathsf{Q}^2}$, kan beräknas efter kvadrering och rotutdragning eller med CORDIC[1]-algoritmen.



Figur 6.1: Insignalen $\vec{v}$ kan delas upp i I- och Q-komposanter.

De rektangulära koordinaterna kan D/A-omvandlas och kopplas till ett xy-kopplat oscilloskop för vidare studium. Den använda omvandlaren MCP4802 har lyckligtvis två utkanaler varför hårdvaran praktiskt taget är förberedd för detta redan.

Kvadratursampling Med kvadratursampling menas att sampla med en frekvens som är fyra gånger högre än den önskade bandbredden. Med $f_s = 7.8$ kHz kan teoretiskt signaler med frekvensinnehåll upp till $f_s/4 \approx 2$ kHz behandlas.

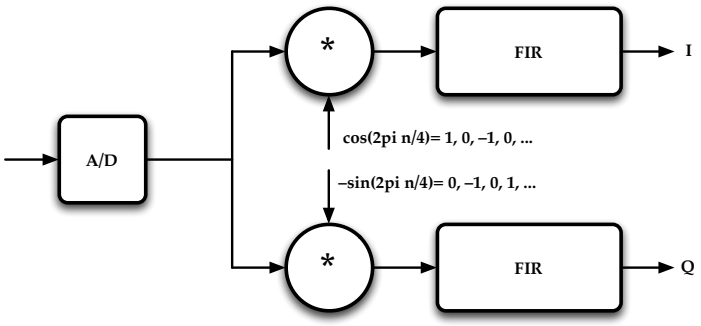
Metoden är en form av korrelering med en cosinus- och en sinusvåg i de punkter dessa är 0 eller ± 1. I praktiken väljer man ut vilka samples som skall adderas $(+1)$, subtraheras (-1) eller inte ingå (0) i signalerna I och Q. Detta tillvägagångssätt medför att

[1] `https://en.wikipedia.org/wiki/CORDIC`

dyra multiplikationer inte behöver genomföras. Däremot måste de korrelerade samplen filtreras/medelvärdesbildas innan de används vidare.

De båda signalerna I- och Q utgör rektangulära koordinater i det komplexa talplanet.

Grafiskt beskrivs kvadratursampling i figuren nedan.



Figur 6.2: Principschema över kvadratursampling. Samplingen utförs med 4x den önskade bandbredden och respektive kvadratursignal filteras innan de behandlas vidare.

Kodmodifieringar Med stegvis förfining som princip måste koden förses med två buffrar (en för I- och en för Q-samplen), två utsignaler (en för vardera filtrerade I- och Q-samplen) och aktivering av den hittills outnyttjade andra D/A-omvandlarkanalen.

`fir` Rutinen modifieras att användas mot valfri buffert. Anropet måste ange mot vilken buffer filtreringen skall ske och vart resultatet skall gå. Här visas koden innan respektive efter de inkrementella förändringarna.

Konstanter och minnesupplägg. Före:

```
        .equ    FIR_SIZE  = 5   ; <-- set this!
        ; --- dont fiddle below. Automatic dependencies!
        .equ    BUFF_SIZE = FIR_SIZE - 1

        .dseg
        .org    SRAM_START
firOut: .byte   1

        .org    $200
Buff:
        .byte   BUFF_SIZE
lastSample:
        .byte   1               ; last sample from A/D
                                ; must be at XH:BUFF_SIZE
```

Konstanter och minnesupplägg. Efter:

```
        .equ    FIR_SIZE  = 5   ; <-- set this!
        ; --- dont fiddle below. Automatic dependencies!
        .equ    BUFF_SIZE = FIR_SIZE - 1
        .dseg

        .org    SRAM_START
OutI:   .byte   1
OutQ:   .byte   1

        ; IN-PHASE buffer
        .org    $200
IBuff:  .byte   BUFF_SIZE
lastSampleI:
        .byte   1               ; last sample from A/D
                                ; must be at XH:BUFF_SIZE
        ; QUADRATURE-PHASE buffer
        .org    $300
QBuff:  .byte   BUFF_SIZE
lastSampleQ:
        .byte   1               ; last sample from A/D
```

Som synes huvudsakligen en dubblering av buffrar och namngivning av minnesplatser *`I` och *`Q`.

Själva filterkoden döptes om till det mer korrekta generiska namnet `fir` och rensades upp något. För att korta ner framställningen nedan har oväsentlig kod ersatts med ":".

FIR-filterkod. Före:

```
fir5:
        push    r18
        push    XL
        clr     r22             ; sum = 0
        clr     r23
        clr     r24
        ldi     ZH,HIGH(firTab*2)
        ldi     ZL,LOW(firTab*2)
        ldi     XL,Buff_Size    ; adress of last (one beyond lastSample)
fir51:
        ld      r16,X           ; lastSample 2's compl
        lpm     r18,Z+          ; r19:r18 16bit 2's compl constant
        lpm     r19,Z+
        :
        brpl    fir1
        sts     firOut,r23
        pop     XL
        pop     r18
        ret
fir5Tab:
        .dw     125,125,125,125,125
```

FIR-filterkod. Efter:

```
fir:
        clr     r22             ; sum = 0
        clr     r23
        clr     r24
        ldi     ZH,HIGH(firTab*2)
        ldi     ZL,LOW(firTab*2)
        ldi     XL,Buff_Size    ; adress of last (one beyond lastSample)
fir1:
        ld      r16,X           ; lastSample 2's compl
        lpm     r18,Z+          ; r19:r18 16bit 2's compl constant
        lpm     r19,Z+
        :
        brpl    fir1
        mov     r16,r23
        ret
firTab:
        .dw     125,125,125,125,125
```

`r16` innehåller nu det beräknade returvärdet.

Anrop av den senare rutinen blir nu någon av nedanstående

```
; ---  filter IN-PHASE
sts     lastSampleI,r16 ; lastSample contains newest sample
ldi     XH,HIGH(IBuff)  ; which buffer to use
call    fir             ; ALWAYS 8-bit result
call    pushBuff        ; push lastSample
sts     OutI,r16        ; 8 bit -128..+127

; ---  filter QUADRATURE-PHASE
sts     lastSampleQ,r16 ; lastSample contains newest sample
ldi     XH,HIGH(QBuff)  ; which buffer to use
call    fir             ; ALWAYS 8-bit result
call    pushBuff        ; push lastSample
sts     OutQ,r16        ; 8 bit -128..+127
```

`pushBuff` är en rutin som skiftar värdena i den utpekade bufferten neråt ett steg och lägger `lastSample` på buffertens nyaste plats. `pushBuff` agerar på den buffer `XH` för tillfället pekar ut.

För att komma till dessa anrop och retur av resultatet genomfördes ett antal små steg vilka var för sig *inte* gjorde koden okörbar:

1. Genomfördes att `fir` kunde arbeta mot den buffer som angavs (`IBuff` eller `QBuff`).
2. Genomfördes att `pushBuff` kunde arbeta mot den buffer som angavs (`IBuff` eller `QBuff`).
3. Genomfördes att `lastSampleI` användes istället för `lastSample`.
4. Genomfördes att resultatet lämnades i `r16` och `sts OutI,r16` lades till efter anropet.
5. Den fungerande `IN-PHASE`-koden kopierades till `QUADRATURE-PHASE`-koden. `lastSampleI` böts mot `lastSampleQ`, samt `OutI` mot `OutQ`.
6. De båda kodstyckena provades sedan var för sig för att avslutningsvis konstatera att båda fungerar som avsett.

Notera att koden kompilerades och provkördes efter *varje* punkt ovan. På detta sätt kunde jag vara säker på att koden hela tiden fungerade.

Och, viktigast, om den plötsligt *inte* fungerat skulle det vara lätt att gå tillbaka och ändra till fungerande kod igen; felet uppstod ju i den senaste modifieringen. Felsökningen[2] eliminieras praktiskt taget med denna metod med små ändringar mellan *fungerande* programversioner.

Slutsatsen är att försöka genomföra så försiktiga kodmodifieringar att koden hela tiden är i ett fungerande skick!

[2] I betydelsen att man går igenom koden och *letar* efter möjliga fel.

Vad som menas med **försiktiga** modifieringar varierar från programmerare till programmerare men behöver inte vara mer än en rad om det känns bekvämast och är möjligt genomförbart.

För D/A-omvandlingen behövdes en rutin för omvandlarens kanal A och en för kanal B. Min försiktiga modifiering av `dac` ovan blev två ingångar, `dacA` och `dacB`, i huvudsakligen samma rutin:

```
        ; --- dacA/B MCP48x2 send r16 (0..FF)
dacA:

        swap    r16
        mov     r17,r16
        andi    r17,$0F
        ori     r17,$10         ;$10=daca
        jmp     dac0
dacB:
        swap    r16
        mov     r17,r16
        andi    r17,$0F
        ori     r17,$90         ; $90=dacb
dac0:
        cbi     PORTB,2         ; SS on device low
        out     SPDR,r17        ; send high byte
dac1:   in      r17,SPSR        ; get status
        sbrs    r17,SPIF        ; wait
        jmp     dac1
        out     SPDR,r16        ; send low byte
dac2:   in      r17,SPSR        ; get status
        sbrs    r17,SPIF        ; wait
        jmp     dac2
        sbi     PORTB,2         ; SS on device high
        ret
```

Det kanske går att komprimera denna rutin mer men då på bekostnad av ökad exekveringstid. Den avvägningen lämnas till senare.

Sammanfattning Rubrikerna hittills sammanfattar de olika stegen i en ingenjörsmässig lösningsgång:

1. Bestäm, åtminstone i grova drag, det slutliga systemets funktion.

   Undvik här att tänka att tänka på *hur* uppgiften skall lösas, tänk mer på det önskade slutresultatet utseende och funktion. Detta steg kan vara en ensidig skiss eller en mer komplett kravspecifikation.

2. Koppla, löd och provkör delarna.

   Utan fungerande delsystem kommer det sammansatta kompletta systemet aldrig kunna fungera. Delsystemen måste konstrueras och testas så de utför de önskade funktionerna.

3. Få ihop något som funkar överhuvudtaget.

   Använd de just konstruerade delsystemen som byggklossar för att bygga ett fungerande system enligt punkt 1.

4. Förfina.

   Här ägnar man rätt mycket tid att konstatera att funktionen verkligen uppfyller kraven. Med fungerande kod kan man faktorisera, modifiera och förbättra funktionen. Ofta utan att funktionen någon gång upphör! Modifiera koden för generalitet och stabilitet.

5. Bygg på funktionalitet.

   Med det fungerande och stabila systemet som grundplåt kan man nu stegvis bygga vidare för utökad funktionalitet.

Alla stegen 1–4 ovan sker i försiktiga och små steg. Något som kanske inte framkommit tydligt är vikten av att kunna mäta resultat ofta, ofta. Efter varje liten kodförändring mättes resultat upp. I detta fall testades också koden kontinuerligt i och med att ljud hela tiden spelades upp i högtalaren. För övrigt användes oscilloskopet precis hela tiden, det startades vid varje arbetspass och användes till passets slut.