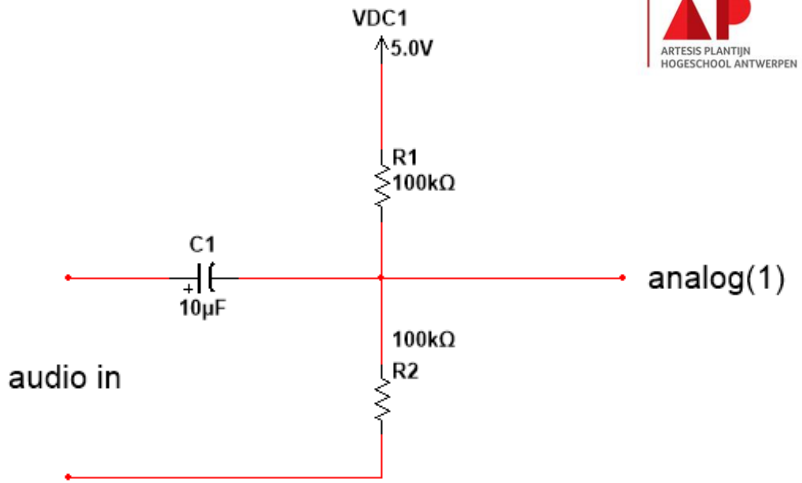
4 – Audio-signaalbewerking

Ing. Patrick Van Houtven

ARTESIS PLANTIJN
HOGESCHOOL ANTWERPEN

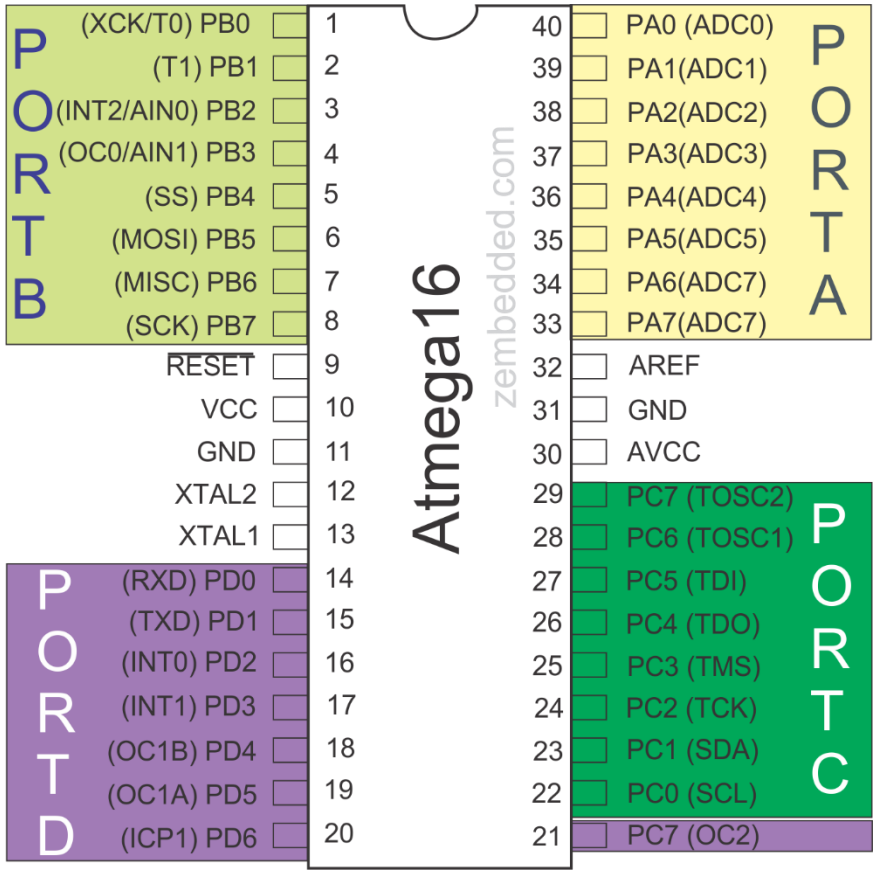
4 Audio-signaalbewerking

- Arduino beschikt over voldoende rekenkracht om audiosignalen in real-time te verwerken
- **Hoe zit het met het audio-ingangssignaal voor Arduino?**
  - Audiosignalen bestaan normaal uit wisselspanningen in bipolaire vorm (plus en min). De analoge ingangen van Arduino kunnen enkel overweg met spanningen tussen 0V en 5V.
  - Hierdoor is het nodig om het audiosignaal te voorzien van een offset van +2,5 V. Dit kan gerealiseerd worden met een spanningsdeler en een condensator waarlangs het signaal wordt toegevoerd.
  - Indien ADC anders wordt ingesteld (interne referentie – externe referentie) dient de 5V voeding aangepast te worden aan de correcte spanningswaarde.
  - De grootte van het ingangssignaal kan aangepast worden met het volume van je PC of ander toestel. Indien nodig kan een voorversterker (bv. Microfoon) gebruikt worden. Ook kan je de ADC in een andere mode plaatsen (bv interne referentie) waardoor het bereik van de ADC wordt aangepast (bv. Van 5V naar 1,1 V)
- **Hoe zit het met aliasing?**
  - Om alliasing te vermijden plaats je best een laagdoorlaatfilter voor de analog(1) pen. Deze heeft een afsnijfrequentie van zo'n 3 kHz.



4-1 Poortregisters

- Zorgen voor een veel snellere toegang en manipulatie van de I/O-pins op het Arduino-bord.
- De chipset voor Arduino beschikt over volgende drie poorten:
  - B (digital pin 8 tot 13
  - C (analog input pins)
  - D (digital pins 0 tot 7)
- Iedere poort wordt gecontroleerd door drie registers die ook als variabelen in de Arduino-taal kunnen worden gedefinieerd.
  - DDR-register : DataDirectionRegister : Bepaald of een bepaalde pin is input of output
  - PORT-register : Controleert of een bepaald pin is HIGH of LOW.
  - PIN-register : leest de status af van de INPUT-pins die als INPUT zijn geplaatst met pinMode().

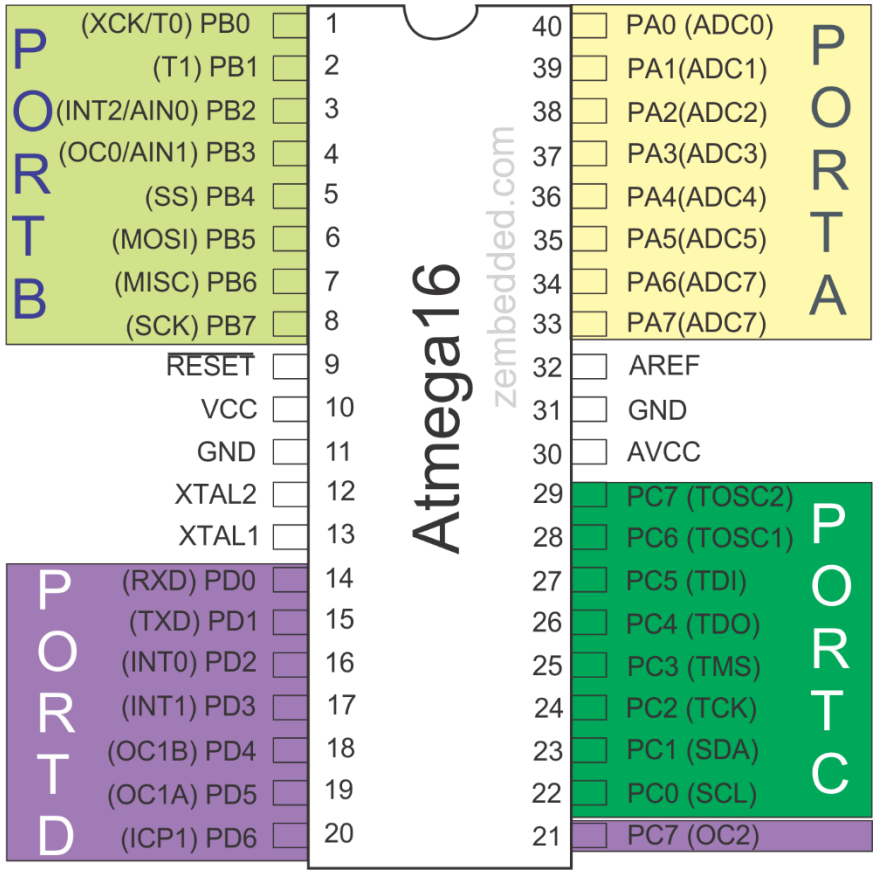


De pinout van Atmega 328 komt overeen met pinout Atmega 168

4-1 Poortregisters

- PORTD mapt naar Arduino digital pins 0 tot 7
  - DDRD : De poort D Data Direction Register – read/write
  - PORTD : De poort D Data Register – read/+write
  - PIND : De poort D inputpins register – read only
- PORTB mapt naar Arduino digital pins 8 tot 13. De twee hogere pinnen (MSB en op één na MSB) mappen naar de kristalpinnen en zijn niet bruikbaar.
  - DDRB : De poort B Data Direction Register – read/write
  - PORTB : De poort B Data Register – read/write
  - PINB : De poort B inputpins register – read only
- PORTC mapt naar Arduino analog pins 0 tot 5. Pinnen 6 en 7 zijn enkel toegankelijk bij de Arduino Mini.
  - DDRC : De poort C Data Direction Register – read/write
  - PORTC : De poort C Data Register – read/+write
  - PINC : De poort C inputpins register – read only
- Ieder pin van bovengenoemde registers komt overeen met een bepaalde pin. Bijvoorbeeld de LSB-bit van DDRB, PORTB en PINB refereren naar PB0 (digital pin 8)

Opmerking : bit 6 en 7 van poort B en poort C worden voor andere zaken gebruikt dan I/O. Wees extra voorzichtig om deze instellingen niet veranderen!



De pinout van Atmega 328 komt overeen met pinout Atmega 168

4-1 Poortregisters

- Voorbeeldcode:
  - Arduinopinnen 0 t/m 7 als OUTPUT plaatsen en pin 0 als INPUT

    DDRD = B11111110;
  - De pinnen 2 t/m 7 als OUTPUT plaatsen zonder de pin-waarden 0 en 1 te beïnvloeden (pin 0 = RX en pin 1 = TX). Dit kan via de OR-functie

    DDRD = DDRD | B11111100;
  - De analoge pinnen 0, 2 en 5 HIGH maken. Opmerking : je gaat enkel 5V op deze pinnen zien als ze als OUTPUT zijn gezet via het DDRC-register of via pinMode().

    PORTC = B00100101;
  - Alle digitale INPUTS tussen pin 8 en pin 13 kunnen in één keer uitgelezen worden via PINB

4-2 Bitwise

- Dikwijls is het interessant om over de mogelijkheid te beschikken om individuele bits te manipuleren. Individuele bitmanipulatie biedt onder andere volgende mogelijkheden:
  - Geheugenbesparing door 8 true/false-waarden op te slaan in één enkele byte.
  - Het aan- en uitzetten van individuele bits in een controleregister of een hardwarepoortregister.
  - Bepaalde bewerkingen uitvoeren aangaande vermenigvuldiging of delen met machten van twee.
- Arduino-compiler biedt geen mogelijkheid om binaire getallen direct uit te drukken in de sourcecode. Wel kunnen er constanten gedefinieerd worden gaande van B0 tot B11111111

4-2 Bitwise

Bitwise AND

- Voorgesteld met een enkele ampersand, &, die gebruikt wordt tussen twee andere integers
- Bitwise AND werkt op iedere bitpositie van de getallen onafhankelijk van elkaar volgens de AND-functie uit de digitale techniek.
- Aldus bekomt men:

  0 & 0 = 0

  0 & 1 = 0

  1 & 0 = 0

  1 & 1 = 1

- In arduino is de type int een 16-bit waarde => & gebruiken tussen twee integers levert 16 simultane AND-operaties op om uit te voeren.
  Voorbeeld :

```
int a = 92; // in binaire vorm :  0000 0000 0101 1100
int b = 101; // in binaire vorm : 0000 0000 0110 0101
int c = a & b; // resultaat :     0000 0000 0100 0100 of 68 decimaal
```

- Meestal wordt bitwise AND gebruikt voor masking. Dit houdt in dat een bepaalde particuliere bit (of bits) van een variabele x wordt geselecteerd en dat deze gestockeerd wordt in een andere variabele y.
  Voorbeeld :

```
int a = 92; // in binaire vorm :  0000 0000 0101 1100
int b = 101; // in binaire vorm : 0000 0000 0110 0101
int c = a & b; // resultaat :     0000 0000 0100 0100 of 68 decimaal

int x = 6; // binaire code : 110
int y = x & 4 // bitwise AND : 110 & 100 => resultaat y = 4 (100)

int x = 7; // binaire code : 111
int y = x & 1 // bitwise AND : 111 & 001 => resultaat y = 1

int x = 5; // binaire code 101
int y = x & 2; // bitwise AND : 101 & 010 => resultaat y = 0
```

Bitwise OR

- Voorgesteld met het vertiaal barsymbool, |, die gebruikt wordt tussen twee integers
- Bitwise OR werkt op iedere bitpositie van de getallen onafhankelijk van elkaar volgens de OR-functie uit de digitale techniek.
- Aldus bekomt men:

  0 | 0 = 0

  0 | 1 = 1

  1 | 0 = 1

  1 | 1 = 1
- In arduino is de type int een 16-bit waarde => | gebruiken tussen twee integers levert 16 simultane OR-operaties op om uit te voeren. Voorbeeld :

```
int a = 92; // in binaire vorm :  0000 0000 0101 1100
int b = 101; // in binaire vorm : 0000 0000 0110 0101
int c = a | b; // resultaat :     0000 0000 0111 1101 of 125 decimaal
```

- Bitwise OR wordt vooral gebruikt om een bepaalde bit op 1 te plaatsen in een bepaalde expressie.
  Voorbeeld :

```
int y = x | 1; de bits van x worden in y gekopieerd maar LSB wordt op 1 gezet

int x = 4; // binaire waarde 100
int y = x | 2; bitwise OR : 100 | 010 => resultaat y = 6 (110)
```

4-2 Bitwise

Bitwise XOR

- Voorgesteld met het caretsymbool, ^, die gebruikt wordt tussen twee integers
- Bitwise XOR werkt op iedere bitpositie van de getallen onafhankelijk van elkaar volgens de XOR-functie uit de digitale techniek.
- Aldus bekomt men:

  0 ^ 0 = 0

  0 ^ 1 = 1

  1 ^ 0 = 1

  1 ^ 1 = 0

- In arduino is de type int een 16-bit waarde => ^ gebruiken tussen twee integers levert 16 simultane XOR-operaties op om uit te voeren. Voorbeeld :

```
int x = 11; // binaire code 1011
int y = 4 ; // binaire code 0100
int z = x ^ y; //bitwise XOR : 1011 ^ 0100 => resultaat : z = 1111 of decimaal 15
```

- Bitwise XOR wordt vooral gebruikt om een bepaalde bit te toggelen (veranderen van 0 naar 1 of van 1 naar 0. Voorbeeld :

```
y = x ^ 1; // LSB van x toggelen (omkeren) en opslaan in y
```

4-2 Bitwise

Bitwise NOT

- Voorgesteld met tildesymbool, ~, die gebruikt wordt tussen twee integers
- Bitwise NOT werkt op iedere bitpositie van de getallen onafhankelijk van elkaar volgens de NOT-functie uit de digitale techniek.
- Aldus bekomt men: 0 wordt 1 en 1 wordt 0.
- In arduino is de type int een 16-bit waarde => ~ gebruiken om de integer te inverteren levert de negatieve waarde van de integer op -1. De waarde wordt opgeslagen in 2-complementnotatie. Hieruit volgt dus: voor iedere integer x geldt dat ~x gelijk is aan –x-1.
- Voorbeeld:

```
int a = 103; //binaire notatie: 0000 0000 1100 0111
int b = ~a; // bitwise NOT :    1111 1111 0011 1000 = -104
```

4-2 Bit shift operators

- Er zijn twee bit shift operators:
  - Shift left operator << : schuift de bits naar links met het aantal opgegeven posities
  - Shift right operator >> : schuift de bits naar rechts met het aantal opgegeven posities
- Voorbeeld:

```
int a = 9; // binair   : 0000 0000 0000 1001
int b = a << 3; binair : 0000 0000 0100 1000  of decimaal 72
int c = b >> 2; binair : 0000 0000 0001 0010  of decimaal 18
```

- Als een waarde x met y posities naar links verschuift, (x << y) verdwijnen de y meest linkse bits.
- Als een waarde x met y posities naar rechts verschuift, (x>>y) verdwijnen de y meest rechtse bits.
- **Vermenigvuldigen met een macht van 2**
  - Schuiven met de shift left operator vermenigvuldigt het getal met een bepaalde macht van 2. Volgende bewerkingen kunnen worden uitgevoerd:

    1 << 0 == 1
    1 << 1 == 2
    1 << 2 == 4
    …
    1 << 8 == 256
    1 << 9 == 512
    1 << 10 == 1024

4-2 Bit shift operators

- **Delen met een macht van 2**
  - Schuiven met de shift right operator vermenigvuldigt het getal met een bepaalde macht van 2. Het resultaat is afhankelijk van wat type variabele je gebruikt.
  - **Type integer**
    - Als x een integer is, is de hoogste bit de tekenbit. Bij een verschuiving naar rechts, treedt sign extenstion op. Dit houdt in dat bij een verschuiving naar rechts, de tekenbit langs rechts wordt ingeschoven. Dit kan leiden tot volgend gedrag:

```
int x = -16;    //binair : 1111 1111 1111 0000
int y = x >>3; // binair : 1111 1111 1111 1110
```

    - Dit is meestal niet wat gewenst is.
  - **Type unsigned integer**
    - Als x een unsigned integer is, is de hoogste bit geen tekenbit. Bij een verschuiving naar rechts wordt er gewoon een 0 ingeschoven langs de rechterzijde.

```
int x = -16;    //binair : 1111 1111 1111 0000
int y = x >>3; // binair : 0001 1111 1111 1110
```

4-4 Assignment Operators

- Resultaten van x (van boven naar onder in vb code

```
int x = 1;    // binair : 0000 0000 0000 0001
x <<= 3;      // binair : 0000 0000 0000 1000  (x = x<<3)
x |= 3;       // binair : 0000 0000 0000 1011  (x = x|3 en 3 = 11)
x &= 1;       // binair : 0000 0000 0000 0001  (x = x|1)
x ^= 4;       // binair : 0000 0000 0000 0101  (toggle via binair mask 100 (=4))
x ^= 4;       // binair : 0000 0000 0000 0001  (toggle via binair mask 100 opnieuw)
```

- Er is geen verkorte notatie voor de bitwise NOT-operator ~. Indien je alle bits wil toggelen kan je dit als volgt doen : x = ~x

4-4 Voorbeelden gebruik poortregisters en bitwise

- **Voorbeeld 1**
  - Stel je wil in je setup()-functie de digitale pinnen 2 tot en met 13 als OUTPUT definieren en vervolgens wil je de pinnen 11, 12 en 13 HIGH maken.
  - Dit kan als volgt:

```
void setup()
    {
        int pin;
        // definiëren van pin 2 t/m 13 als OUTPUT
        for (pin=2; pin <= 13; ++pin) {
            pinMode (pin, OUTPUT);
        }
        // laag maken van pin 2 t/m 10
        for (pin=2; pin <= 10; ++pin) {
            digitalWrite (pin, LOW);
        }
        // hoog maken van pin 11 t/m 13
        for (pin=11; pin <= 13; ++pin) {
            digitalWrite (pin, HIGH);
        }
    }
```

Dezelfde instelling door gebruik te maken van de hardware poorten en bitwise operatoren:

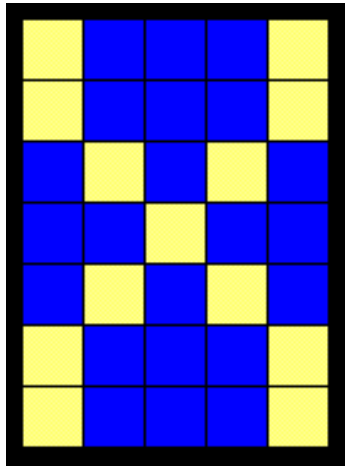
```
void setup()
    {
        // Maak pin 1(serial transmit) en pinnen 2 t/m 7 als OUTPUT
        // Pin0 (serial receive) moet ingesteld blijven als INPUT
        // omdat anders de seriële poort stopt met werken
        DDRD = B11111110;  // digitale pins 7,6,5,4,3,2,1 = OUTPUT
                           // en pin 0 = INPUT

        // Maak pin 8 t/m 13 als OUTPUT
        // pin 14 en 15 zijn kristalpinnen en dus niet bruikbaar voor OUTPUT
        DDRB = B00111111;  // digitae pins -,-,13,12,11,10,9,8 als OUTPUT

        // Enkel pin 11 t/m 13 HIGH maken en de andere pins 2 t/m10 LOW maken
        // van pin 0 en 1 wordt afgebleven (pin 0 = RXD en pin 1 = TXD)
        PORTD &= B00000011;   // LOW maken pin 2 t/m 7 en pin 0
                              // en pin 1 veranderen niet door de bitwise AND

        // Gelijktijdig de pin 11 t/m 13 HIGH maken
        PORTB = B00111000;    // Hiermee zijn pin 13,12 en 11 hoog gemaakt
                              // en pin 10, 9, en 8 laag gemaakt
    }
```

4-4 Voorbeelden gebruik poortregisters en bitwise

- **Voorbeeld 1** : besluiten
  - Stel je wil in je setup()-functie de digitale pinnen 2 tot en met 13 als OUTPUT definiëren en vervolgens wil je de pinnen 11, 12 en 13 HIGH maken.
  - **Werken met digital.Write() en digitalRead() biedt volgende voordelen :**
    - De code is gemakkelijker te lezen
    - Gebruik van digital.Write en digital.Read kan op alle Atmel µC's gebruikt worden terwijl control- en poort-registers kunnen verschillen naargelang de microcontroller
  - **Werken met control- en poortregister biedt volgende voordelen:**
    - Bij programma's die veel geheugen vereisen biedt deze de mogelijkheid om de code te verkleinen
    - Je kan gelijktijdig meerdere pinnen schrijven in plaats van achter elkaar => veel snellere uitvoering vermits digital.Write en digital.Read veel onderliggende code in machinecycli bevat

4-4 Voorbeelden gebruik poortregisters en bitwise

- **Voorbeeld 2** :
  - Stel je bouwt je eigen LED-grid waarop je bepaalde symbolen op wil weergeven door individuele LED's aan of uit te zetten. Stel een grid van 5-7 bitmap voor de letter X als volgt:



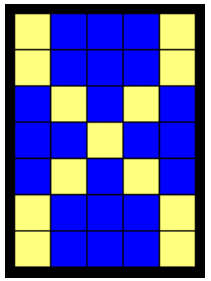
```
const prog_uint8_t BitMap[5][7] = {   // bewaar in het programmageheugen om RAM uit te sparen
        {1,1,0,0,0,1,1},
        {0,0,1,0,1,0,0},
        {0,0,0,1,0,0,0},
        {0,0,1,0,1,0,0},
        {1,1,0,0,0,1,1}
    };

    void DisplayBitMap()
    {
        for (byte x=0; x<5; ++x) {
            for (byte y=0; y<7; ++y) {
                byte data = pgm_read_byte (&BitMap[x][y]);   // fetch data from program memory
                if (data) {
                    // turn on the LED at location (x,y)
                } else {
                    // turn off the LED at location (x,y)
                }
            }
        }
    }
```

  - Een gemakkelijke manier om zo'n beeld te bewaren is gebruik maken van een array bestaande uit integers. De code hiervoor is hiernaast afgebeeld.
  - prog_uint8_t is een speciaal gedefinieerd type (pgmspace.h) waarmee data in het flashgeheugen kan worden geplaatst door er arrays in the definiëren

- Iedere pixel in de bitmap gebruikt 1 byte programmageheugen. De bitmap gebruikt bijgevolg 5x7 = 35 bytes geheugen.
- Indien voor ieder karakter van de ASCII-characterset zo'n bitmap moet worden gemaakt, neemt dit 96 x 5 x7 = 3360 bytes in beslag (3,3 kb)

4-4 Voorbeelden gebruik poortregisters en bitwise

- **Voorbeeld 2** :
  - Stel je bouwt je eigen LED-grid waarop je bepaalde symbolen op wil weergeven door individuele LED's aan of uit te zetten. Stel een grid van 5-7 bitmap voor de letter X als volgt:



```
const prog_uint8_t BitMap[5] = {
  // store in program memory to save RAM
        B1100011,
        B0010100,
        B0001000,
        B0010100,
        B1100011
    };
```

  - Een meer efficiëntere manier bestaat erin de 2-dimensionale array te vervangen door een 1-dimensionale array bestaande uit bytes.
  - Iedere byte bevat 8 bits waarvan de 7 laagste bits gebruikt worden om de 7 pixels van een kolom in de bitmap weer te geven.
  - Er is gebruik gemaakt van de voorgedefinieerde binary constanten die beschikbaar zijn vanaf Arduino 0007. Hierdoor hoeven we slechts 5 bytes te gebruiken in plaats van 35.
  - Om gebruik te kunnen maken van deze meer compacte data wordt de functie DisplayBitMap() omgevormd tot een functie die toegang tot deze meer compacte dataformaat toelaat:

```
void DisplayBitMap()
    {
        for (byte x=0; x<5; ++x) {
            byte data = pgm_read_byte (&BitMap[x]);   // fetch data from program memory
            for (byte y=0; y<7; ++y) {
                if (data & (1<<y)) {
                    // turn on the LED at location (x,y)
                } else {
                    // turn off the LED at location (x,y)
                }
            }
        }
    }
```

  - pgm_read_byte() is een macro dat een byte leest die opgeslagen is in een gespecificeerd adres in de PROGMEM-area. Deze macro is gedefinieerd in hardware/tools/avr/avr/include/avr/pgmspace.h
  - &BitMap[x] binnen pgm_read_byte (&BitMap[x]) stelt een pointer voor naar de locatie waar de BitMap in het PROGMEM is opgeslagen

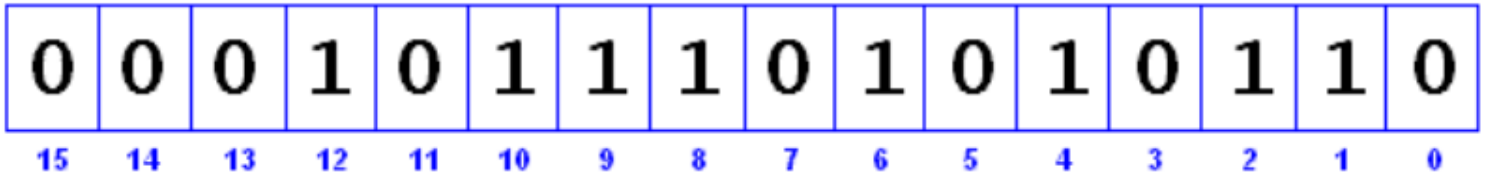
4-4 Voorbeelden gebruik poortregisters

- if (data & (1<<y)) {
  - DE expressie (1<<y) selecteert een gegeven bit binnen in de data waar we toegang tot willen hebben
  - De bitwise AND-functie data & (1<<y) test een gegeven bit.
    - Als deze bit de waarde 1 heeft (set), een nonzerowaarde is de uitkomst.
    - Als deze bit de waarde 0 heeft, is de uitkomst zero
  - De if die voor (data & (1<<y) staat zorgt ervoor dat als de bitwise AND als resultaat 1 geeft, het resultaat als zijnde TRUE wordt aanzien. In het ander geval levert dit FALSE op zodat else wordt uitgevoerd.

```
const prog_uint8_t BitMap[5] = {
  // store in program memory to save RAM
        B1100011,
        B0010100,
        B0001000,
        B0010100,
        B1100011
    };
    void DisplayBitMap()
    {
        for (byte x=0; x<5; ++x) {
            byte data = pgm_read_byte (&BitMap[x]);   // fetch data from program memory
            for (byte y=0; y<7; ++y) {
                if (data & (1<<y)) {
                    // turn on the LED at location (x,y)
                } else {
                    // turn off the LED at location (x,y)
                }
            }
        }
    }
```

4-4 Voorbeelden gebruik poortregisters en bitwise

- **Voorbeeld 3** :
  - Snel refereren naar een bepaalde bit in een 16-bit integer waarbij gestart wordt vanaf LSB (bit 0) en waarbij bit 15 gelijk is aan de tekenbit bij signed integer of gewoon bit 15 bij unsigned integer. Beschouw onderstaand voorbeeld:



- De variabele n geeft de bitpositie aan binnen de integer (n heeft dus een waarde tussen 0 en 15)

- y = (x >>n) &1; // bewaard de n^{de} bit van x in y; y wordt 0 of 1
- x &= ~(1 << n); // forceert de n^{de} bit van x naar 0 terwijl al de andere bits hun waarde behouden
- x &= (1 << (n+1))-1; // laat de laagste n bits van x zoals ze zijn en plaatst al de hogere bits op 0
- x |= (1 <<n); // forceert de n^{de} bit van x om 1 te worden en laat de andere bits hun waarde behouden
- x ^= (1<<n); // toggelt de n^{de} bit van x, al de andere bits behouden hun waarde
- x = ~x; // toggelt alle bits van x

4-4 opgaven

Opgave 1:
Schrijf een sketch waarbij je een trapspanningsgenerator maakt met arduino en een somversterker.
De trapspanningsgenerator vertrekt vanaf 0V DC en gaat per seconde met 0,5 V omhoog tot de spanning van 10 V wordt bereikt. Vervolgens daalt de spanning telkens met een halve volt tot 0V wordt bereikt enz …
Het is de bedoeling dat je telkens in 1 keer naar de 4 pinnen van poort C een digitale code schrijft.

Opgave 2:
Volgende sketch stelt een VCO voor.
De signaalfrequentie wordt ingesteld met de spanning op A0. Deze spanning wordt bekomen via een potentiometer van 10 kΩ.
Het frequentiebereik loopt van 30 Hz tot (meer dan) 5 kHz.
In de sketch zijn 2 functies aanwezig.

- ***void calcSine()***
  - ***Hierin wordt een tabel met sinuswaarden berekend (eenmalige berekening die dan wordt opgeslagen)***
- ***ISR(TIMER2_OVF_vect)***
  - ***Wordt elke 16 ms aangeroepen via een automatische interruptfunctie die met een frequentie van 62,5 kHz optreedt.***
  - ***Bij elke vierde aanroep van deze interruptfunctie wordt de spanningswaarde op de analoge ingang A0 ingelezen en in de hoofdprogrammalus gebruikt om de frequentie van de sinus in te stellen.***

- ***setup()***
  - ***Instellen van Timer2 zadat deze in de Fast PWM-mode werkt. Deze genereert een automatische interrupt . Dit wordt gedaan via het rechtstreeks adresseren van een specifiek register van de Atmega 328.***
- ***Met sbi kan je een bit setten en met cbi kan je een bit clearen.***
- ***Maak deze VCO en voer deze uit. Analyseer het programma, verklaar het werkingsprincipe en zet dit om in een flowchart.***

4-4 opgaven

```
#define sbi(PORT, bit) (PORT |= 1 << bit);
#define cbi(PORT, bit) (PORT &= ~(1 << bit));

const int pwmOutPin = 11; // PWM uitgang
const int testPin = 3;
const float pi = 3.141592;

int indexCnt, interCnt; // index - interrupt-tellers
byte sine[512]; // array voor sinuswaarden

// variabelen voor interruptbesturing
volatile boolean adcSample;
volatile byte bufferAdc0;
```

```
void setup() {
  pinMode(pwmOutPin, OUTPUT); //PWM-pen als uitgang
  pinMode(testPin, OUTPUT); //testpen als uitgang
  calcSine(); // sinustabel inladen in geheugen

  //stel ADC-prescaler in op 64, 8-bit ADC en VCC-referentie
  //selecteer analoog kanaal 0
  ADCSRA |= 0b00000011;
  ADMUX |= 0b11000000;

  //instellen Timer2 op fast PWM-modus
  TCCR2A |= 0b10000011;

  // instellen Timer2 geen prescaler
  TCCR2B |= 0b00000001;
  cbi(TCCR2B, CS21);
  cbi(TCCR2B, CS22);

  // uitschakelen Timer0 en activeren Timer2 interrupt
  cbi(TIMSK0, TOIE0);
  sbi(TIMSK2, TOIE2);

}
```

4-4 opgaven

```
void loop() {
  while (!adcSample) {} // wacht op nieuwe ADC-waarden
  digitalWrite(testPin, HIGH);
  adcSample = false;
  OCR2A = sine[indexCnt]; //stuur huidige waarde naar PWM
  indexCnt++;
  indexCnt +=bufferAdc0;
  indexCnt &= 511;
  digitalWrite(testPin, LOW);
}
```

```
// bereken de sinus-array met 512 waarden tussen 0 en 2 * pi
void calcSine() {
  for (int n = 0; n << 511; n++) {
    sine[n] = 127 * sin(pi * n /255) + 128;
  }
 }

 // haal potmeter-waarde op
 ISR(TIMER2_OVF_vect) {
   interCnt++; // reduceer samplingtempo van 62.5 kHz naar 15.625 kHz
   if (4 == interCnt) {
   bufferAdc0 = ADCH; // haal 8-bit ADC-waarde op
   adcSample = true;
   sbi(ADCSRA, ADSC); // begin volgende omzetting
   interCnt = 0;
   }
 }
```

Macro's

- **Hoe macro opbouwen?**

  #define XXXXX YYYYY

  - Met XXXXX de macronaam die je wil gebruiken voor je macro. Voor een beter onderscheid in je code kan het handig zijn hiervoor hoofletters te gebruiken.
  - Met YYYYY de code die je wil toevoegen voordat gecompileerd wordt.
  - Voorbeeld :

    #define TOGGLEd2 PORTD ^= B00000100

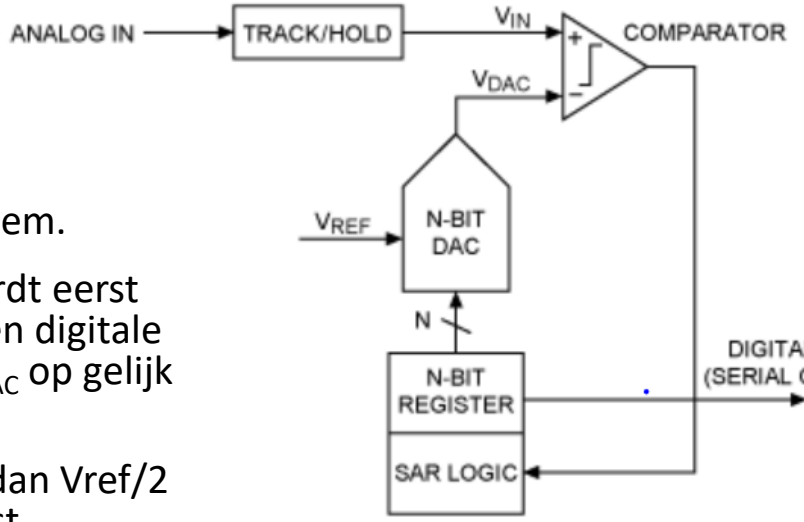
    - TOGGLEd2 = is de naam voor de macro
    - PORTD ^= B00000100 = de code die de microcontroller moet uitvoeren

- **Macro sbi**
  - Doel : Het zetten van een bepaalde bit op 1 in een bepaalde poort.
  - Code : #define sbi(PORT, bit) (PORT |= 1 << bit);

- **Macro cbi**
  - Doel het plaatsen van een bepaalde bit op 0 in een bepaalde poort.
  - Code : #define cbi(PORT, bit) (PORT &= ~(1 << bit));

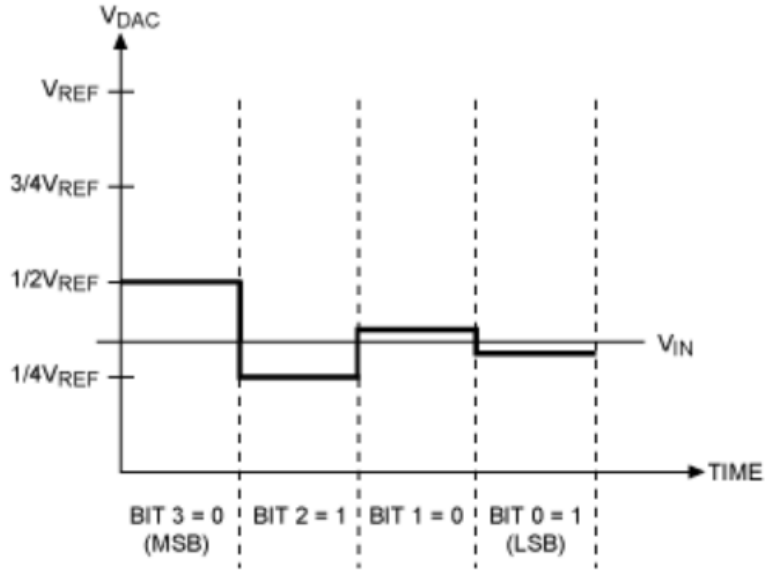
4-6 ADC

- Er bestaan verschillende architecturen om een ADC op te bouwen. Alle AVRs dat een ADC bevatten werken volgens de successive approximation methode.



Principe successive approximation methode

- De analoge ingangsspanning (V_{IN}) wordt vastgehouden via het track/hold systeem.
- Om de overeenkomstige digitale code te vinden van deze ingangsspanning wordt eerst de MSB van het N-bit register hoog gemaakt en de overige bits 0. Dit levert een digitale code op gelijk aan Vref/2. Via de N-bit DAC levert dit een analoge spanning V_{DAC} op gelijk aan Vref/2.
- Via de comparator wordt deze spanning vergeleken met V_{IN} . Als V_{IN} groter is dan Vref/2 dan blijft de MSB-bit op 1 staan . In het andere geval wordt deze op 0 geplaatst.
- Als de MSB-bit afgehandeld is, wordt de volgende bit op 1 gezet en wordt er terug vergeleken. Is V_{IN} groter dan V_{DAC} dan zal deze bit op 1 blijven, in het andere geval wordt deze terug gereset.
- Daarna is de volgende bit aan beurt. Dit blijft zo doorgaan tot de LSB-bit is afgehandeld. De code die dan in het N-bit register staat is de quantisatiewaarde van V_{IN}.

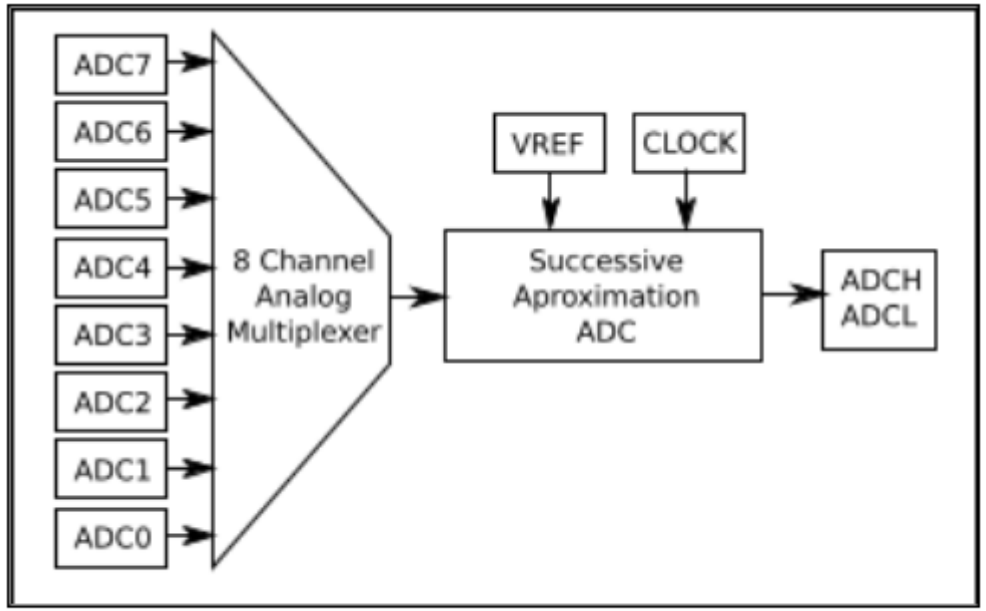


- **Voorbeeld : 4-bit ADC volgens successive approximation methode**
  - Bit 3 : $V_{DAC} > V_{IN}$: waardoor Bit3= 0
  - Bit 2 : $V_{DAC} < V_{IN}$: waardoor Bit2= 1
  - Bit 1 : $V_{DAC} > V_{IN}$: waardoor Bit2= 0
  - Bit 0 : $V_{DAC} < V_{IN}$: waardoor Bit0= 1

4-6 ADC

Arduino-ADC

- Arduino beschikt over een 8-kanaals analoge multiplexer welke een ADC bevat.
  - Multiplext de 8 analoge pinnen naar één 10-bit ADC
  - Enkel 1 ADC-operatie kan per keer worden verwerkt. Indien meer dan 1 ADC-pin gebruikt wordt, zullen de uitlezingen 1 per 1 gebeuren. Voorbeeld : stel dat analog pin 0 wordt geconverteerd, dan kan analog pin 1 pas geconverteerd worden nadat de quantisatie voor analog 0 is afgerond.
- Arduino Uno beschikt over een 8-bit microcontroller en de ADC heeft een 10-bit resolutie. Hierdoor wordt het resultaat van de ADC-conversie opgeslagen in twee registers die ADCH en ADCL worden genoemd (ADC High en ADC Low)
- De input V_{IN} moet liggen tussen 0V en Vref. Meestal is Vref gelijk aan de voedingsspanning welke voor de meeste Arduinobordjes 5 V is.
- V_{IN} is een spanning die afkomstig is van een bepaalde sensor (bv. temperatuur) en wordt aangelegd aan een bepaalde ADC-kanaal (analog input pin)
- De formule om een ADC-conversie te berekenen : $(V_{IN}/Vref)*1023$



ADC

4-6 ADC

Operating modes ADC

- 2 operating modes : single conversion en free running

Single Conversion Mode

- In deze mode moet je elke conversie starten.
- Als omzetting is voltooid, worden de resultaten in **ADCH**- en **ADCL**-registers geplaatst en ADIF is geset. Er wordt geen andere conversie gestart.
- Een conversie wordt ingesteld door de **ADSC** (ADC start conversie) bit in het ADCSR-register (**ADC C**ontrol en **S**tatus **R**egister **A**) register te setten. ADSC blijft hoog gedurende de ganse conversie en zal automatisch worden gewist als de conversie ten einde is.
- Je kan kiezen welk kanaal je wil converteren via het **ADMUX** register voordat de conversie start. Vermits je met 3 bits 8 kanalen kan selecteren zijn in dit register slechts 3 bits (de 3 meest LSB-bits) hiervoor in gebruik. Op die wijze kan je het gewenste kanaal rechtstreeks naar het register schrijven zonder andere vlaggen of bitsettingen te beïnvloeden. Als deze bits veranderen tijdens een conversie, heeft deze verandering geen effect vooraleer de conversie is beëindigd.
- Melding dat de conversie voltooid is kan bekomen worden via het ADC Conversion Complete Interrupt service routine (**ISR**). Als je ervoor zorgt dat **ISR** steeds de **ADMUX**-waarde verandert voor het volgende kanaal (of naar een bepaald kanaal), dan is de waarde in het ADC-dataregisterpaar (ADCH en ADCL) steeds het conversieresulstaat van de laatste ADCMUX verandering

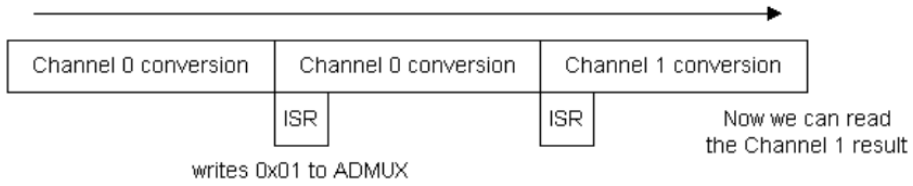
ADCSR

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADMUX

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
---	---	---	---	---	MUX2	MUX1	MUX0

MUX2	MUX1	MUX0	Selected Input
0	0	0	ADC0
0	0	1	ADC1
0	1	0	ADC2
0	1	1	ADC3
1	0	0	ADC4
1	0	1	ADC5
1	1	0	ADC6
1	1	1	ADC7



Free Running mode

- In deze mode initialiseer je de eerste conversie en dan zal de ADC automatisch de volgende conversies starten van zodra de voorgaande conversie afgehandeld is.
- Welk kanaal je wil converteren geef je aan via de instelling van het ADMUX-register.
- Als een nieuw kanaal moet geselecteerd worden, dan moet dit in het ADMUX-register aangegeven zijn vooraleer de volgende conversie begint.
- Als je gebruik maakt van ISR voor het verkrijgen van de ADC-resultaten en het selecteren van het ADC-kanaal, let dan op dat je het kanaal niet verandert op het moment dat de conversie juist is gestart. Direct veranderen van het kanaal nadat de conversie juist is gestart kan leiden tot onvoorspelbaar gedrag van de ADC.

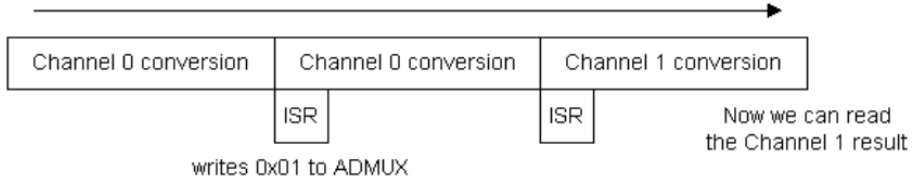
ADCSR

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADMUX

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
---	---	---	---	---	MUX2	MUX1	MUX0

MUX2	MUX1	MUX0	Selected Input
0	0	0	ADC0
0	0	1	ADC1
0	1	0	ADC2
0	1	1	ADC3
1	0	0	ADC4
1	0	1	ADC5
1	1	0	ADC6
1	1	1	ADC7



Registers

- Er zijn 4 registers betrokken bij de werking van de ADC.
  - ADC-mulitplexer select register (ADCMUX)
  - ADC Control and Status Register (ADCSR)
  - ADC Data Register Low (ADCL)
  - ADC Data Register High (ADCH)

ADCSR

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

ADSR

- ADEN (ADC Enable) bit
  - ADEN = 1 enables de ADC
  - ADEN = 0 zet de ADC af, ADEN gelijk maken aan 0 terwijl een conversie nog bezig is zal deze conversie onmiddellijk beëindigen.
- ADSC (ADC Start Conversion) bit
  - Free running mode : je moet deze bit op 1 zetten om de eerste conversie te starten. De volgende conversies starten dan automatisch
  - Single Conversie Mode : je moet deze bit op 1 zetten voor iedere conversie die je wil uitvoeren. Deze bit wordt automatisch op 0 gezet door hardware telkens een conversie is uitgevoerd.
  - Opgelet! De eerste conversie die uitgevoerd wordt nadat de ADC is enabled is een extended conversion en een extended conversion zal de ADSC-bit niet clearen na voltooing van een extended conversion.
- ADFR (ADC Free Running Select) bit
  - ADFR = 1 als je de ADC wil gebruiken in de free running mode
  - ADFR = 0 als je de ADC wil gebruiken in de single conversion mode

ADCSR

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

- ADIF (ADC Interrupt Flag) bit
  - Deze bit is gelijk aan 1 als de conversie beëindigd is. Als dan ook ADIE (ADC interrupt enabled) gelijk is aan 1 en globale interrups zijn enabled wordt de ADC Conversion Complete Interrupt uitgevoerd.
  - ADIF wordt gewist door hardware tijdens het uitvoeren van de corresponderende interrupt handling vector
  - ADIF kan eveneens worden gewist door een logisch 1 te schrijven naar deze vlag (bitpositie 4)
  - Wanneer je wijzigingen aanbrengt aan een bepaalde bit van ADCSR door gebruik te maken van de sbi- of cbi-instructie zal ADIF gewist worden (cleared) als deze op 1 was gezet voordat de sbi- of cbi-instructie wordt uitgevoerd.
- ADIE (ADC Interrupt Enable) bit
  - ADIE = 1 en global interrupts is enabled => ADC-interrupt is geactiveerd en de ADC interrupt routine wordt opgeroepen nadat een conversie is afgerond.
  - ADIE = 0 => de interrupt is disabled
- ADPS (ADC Prescaler) bits
  - Deze bits bepalen de deelfactor tussen de AVR-klokfrequentie en de ADC klokfrequentie.
- ADCL en ADCH
  - Bevatten het resultaat van de laatste conversie.
    - ADCH bevat de twee meest MSB-bits van de conversie en ADCL de overige 8 bits.
    - Als ADCL wordt gelezen, worden de registers niet geupdaded zolang ADCH niet is uitgelezen
    - Het is belangrijk dat beide registers worden uitgelezen en dat ADCL wordt uitgelezen voor ADCH

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

4-6 ADC

ADCSR

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Kloksnelheid

- De ADC met 10-bits resolutie heeft een aanbevolen kloksnelheid tussen 50 kHz en 200 kHz. Hogere snelheden zorgen er voor dat de resolutie zal zakken.
- Met de prescaler kan de ADC-klok onder controle gebracht worden. Een arduino met kloksnelheid 16 MHz kan de kloksnelheid voor de ADC op volgende waarden ingesteld worden:
  - 16 MHz /2 = 8 MHz
  - 16 MHz /4 = 4 MHz
  - 16 MHz /8 = 2 MHz
  - 16 MHz /16 = 1 MHz
  - 16 MHz /36 = 500 kHz
  - 16 MHz /64 = 250 kHz
  - 16 MHz /128 = 125 kHz

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

- Andere kloksnelheden kunnen bekomen worden door kloksnelheid van het bordje te veranderen. . Bijvoorbeeld 12 MHZ. Delen door 4 levert 3 MHz op, enz…
- Een normale conversie in de ADC neemt 13 ADC-klokpulsen in beslag. De eerste conversie neemt 25 klokpulsen in beslag. Dit houdt in dat de ADC-kloksnelheid nog gedeeld moet worden door 13 om te weten te komen hoeveel samples er per seconden kunnen verwerkt worden.
  - De arduino library (file wiring.c waarin de ADC wordt geconfigureerd) stelt de ADC standaard in met een 128 prescaler. Dit levert een 125 kHz ADC-klok op als het bordje werkt met een kloksnelheid van 16 MHz.
  - Met een ADC-kloksnelheid van 125 kHz levert dit 125 kHz/13 = 9600 Hz samplesnelheid op. Dit betekent dat de hardwarelimiet bij 10-bit resolutie ligt op 9600 samples per seconde of 96 samples per miliseconde.
  - Als de ADC-kloksnelheid op zijn maximale snelheid wordt ingesteld (200 kHz) dan resulteert dit in 200 kHz/13 = 15384 samples per seconde. Dat is het beste wat kan worden bekomen met de Atmega328P ADC op 10-bit resolutie.

4-6 ADC

Resolutie

- De ADC heeft een 10-bit resolutie. Hiermee worden waarden tussen 0 en 1023 bekomen.
  - Als referentiespanning gelijk is aan 5 V, dan is de kleinst detecteerbare verandering in spanning gelijk aan 0,0049 V of 4,9 mV
- De ADC heeft een aanbevolen maximum ADC kloksnelheid van 200 kHz.

Opgave 3:
Ga na met naaststaande sketch waarmee aan de hand van 100 analoge leeswaarden van de ADC de verwerkingstijd van iedere leeswaarde wordt bepaald. De resultaten worden uitgeprint in de seriële console. Als analog pin wordt pin 2 gebruikt waar een potentiometer is verbonden om het ganse bereik van de ADC weer te kunnen geven.

```
1  // Arrays om de resultaten in te bewaren
2  unsigned long start_tijden[100];
3  unsigned long stop_tijden[100];
4  unsigned long waarden[100];
5
6  // Setup  seriële poort en pin 2
7  void setup() {
8    Serial.begin(9600);
9    pinMode(2, INPUT);
10 }
11 void loop() {
12   unsigned int i;
13
14   // vastleggen van de waarden in het geheugen
15   for(i=0;i<100;i++) {
16     start_tijden[i] = micros();
17     waarden[i] = analogRead(2);
18     stop_tijden[i] = micros();
19   }
20
21   // afdrukken van de resultaten
22   Serial.println("\n\n--- Resultaten ---");
23   for(i=0;i<100;i++) {
24     Serial.print(waarden[i]);
25     Serial.print(" verloop = ");
26     Serial.print(stop_tijden[i] - start_tijden[i]);
27     Serial.print(" us\n");
28 }
29   delay(6000);
30 }
```

4-6 ADC

Opgave 4:
Pas de ADC-prescaler aan zoals links weergegeven.
Ga na wat de verwerkingstijden zijn voor iedere instelling en verklaar de werking van de sketch. Doe dit aan de hand van een flowchart (stroomdiagram)

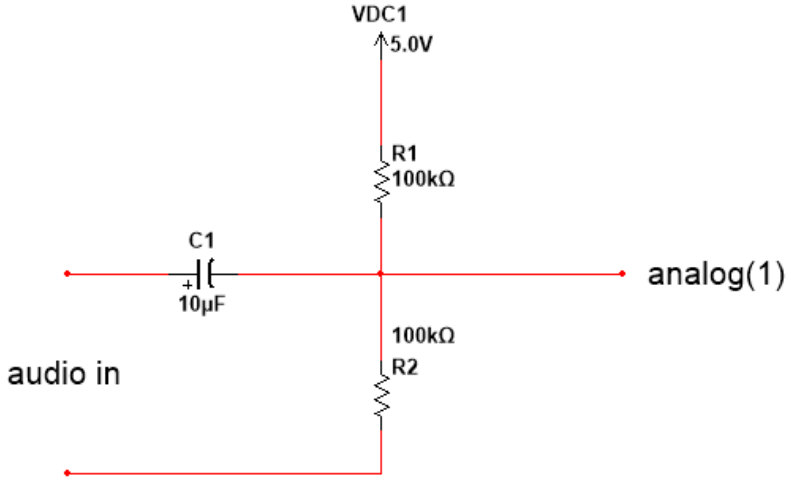
Wat zijn je conclusies aangaande resolutie (nauwkeurigheid van het signaal) als de prescaler verandert wordt?

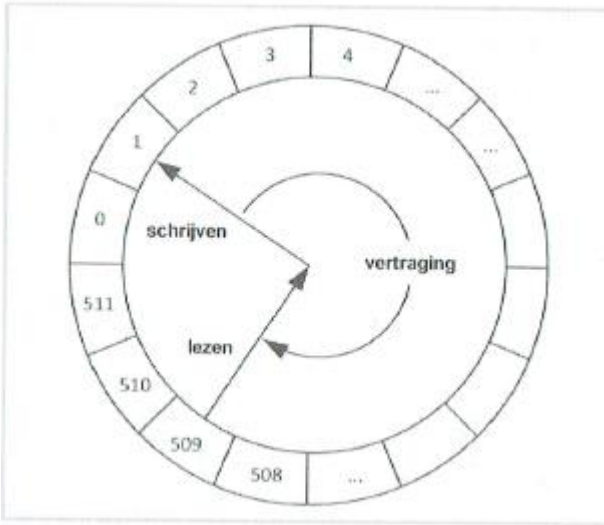
Opgave 5
Pas de sketch aan zodat je ook de conversiewaarden zichtbaar maakt.
Stel de ingangsspanning op pen 2 achtereenvolgens in op 1 V, 2,5 V en 4 V. Ga na hoeveel de conversiewaarde bedraagt per prescalerinstelling.
Stel een tabel samen waarin per prescalerinstelling de sampletijd en conversiewaarde wordt weergegeven. In hoeverre verandert de nauwkeurigheid van de ADC bij de verschillende prescalerinstellingen?

```
// Arrays om de resultaten in te bewaren
unsigned long start_tijden[100];
unsigned long stop_tijden[100];
unsigned long waarden[100];

// definieren van verschillende ADC-prescalers
const unsigned char PS_16 = (1 << ADPS2);
const unsigned char PS_32 = (1 << ADPS2) | (1 << ADPS0);
const unsigned char PS_64 = (1 << ADPS2) | (1 << ADPS1);
const unsigned char PS_128 = (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);


// Setup  seriële poort en pin 2
void setup() {
  Serial.begin(9600);
  pinMode(2, INPUT);

 // instellen van de ADC
  ADCSRA &= ~PS_128;  // verwijder de instellingen van de Arduinobibliotheek

  // Kiezen van 1 van de prescalerinstellingen hierboven gedefinieerd
  // PS_16, PS_32, PS_64 of PS_128
  ADCSRA |= PS_64;    // prescaler instellen op 64


}
void loop() {
  unsigned int i;

  // vastleggen van de waarden in het geheugen
  for(i=0;i<100;i++) {
    start_tijden[i] = micros();
    waarden[i] = analogRead(2);
    stop_tijden[i] = micros();
  }

  // afdrukken van de resultaten
  Serial.println("\n\n--- Resultaten ---");
  for(i=0;i<100;i++) {
    Serial.print(waarden[i]);
    Serial.print(" verloop = ");
    Serial.print(stop_tijden[i] - start_tijden[i]);
    Serial.print(" us\n");
}
  delay(6000);
}
```

4-7 Genereren van nagalmeffect

- ***Principe: door gebruik te maken van een circulair buffergeheugen***.
  - Analoog signaal wordt eerst gedigitaliseerd met AD-convertor.
    - Vermits analoge signaal uit wisselspanningen bestaat is er aan de ingang van de ADC een offsettspanning van + 2,5 V noodzakelijk.
  - De gedigitaliseerde waarden worden vervolgens in een array opgeslagen.
    - Tengevolge van de beperkte systeembronnen van de Atmega328 is de omvang van het array beperkt tot 512 bytes
    - De array wordt gebruikt als circulaire buffer en om de vertraging te realiseren (die noodzakelijk is voor het nagalmeffect) wordt deze buffer herhaaldelijk uitgelezen.
  - De vertraging wordt ingesteld met een afzonderlijke potentiometer.
    - In deze toepassing wordt zowel de potentiometerspanning als het analoog ingangssignaal door de interruptroutine bemonsterd.
    - Hiervoor worden analog input 1 (analoog signaal) en analog input 2 (potentiometer) om beurten ingelezen.
  - Het galmsignaal wordt naar buiten gebracht via de Fast PWM-modus.
  - In het hoofdprogramma worden de waarden in de circulaire buffer geschreven en met enige vertraging weer uitgelezen.





4-7 Genereren van nagalmeffect

- Macro’s sbi en cbi worden gebruikt om bepaalde bits te setten of te resetten in een aantal registers
- Pin 11 wordt als uitgang gebruikt voor het galmsignaal
- Volatile (vluchtig) geeft aan dat de waarde kan wisselen tussen de verschillende toegangen tot de variabele, zelfs als het lijkt niet te worden gewijzigd. Met volatile wordt voorkomen dat de compiler een optimalisatie doorvoert en de variabele als een constante (ogenschijnlijk niet wijzigbaar) gaat aanzien.
- Volatile wordt voornamelijk gebruikt bij hardware access (memory-mapped I / O), waarbij het lezen van- of schrijven naar het geheugen wordt gebruikt om te communiceren met randapparatuur.

```
1  // echo produceren met arduino
2
3  //definieren macro's
4  #define cbi(PORT, bit) (PORT &= ~(1 << bit))
5  #define sbi(PORT, bit) (PORT |= 1 << bit)
6
7  const int pwmOUTPin = 11; // PWM-uitgang
8  const float pi = 3.1415926535;
9  const byte halfMax = 127;
10 boolean div2, selectChannel; // stuurvariabele voor sampling
11
12 // globaal toegankelijke interruptvariabelen
13 volatile boolean adcSample;
14 volatile byte potSample, analogSample;
15 int indexCounter, valAnalog, valPot;
16 byte pwmBuf, ringBuff[512]; //audio ringbuffer-array 8-bit
17
```

4-7 Genereren van nagalmeffect

- ADCSRA = ADC Control en Status Register A
  - Met ADPS2, ADPS1 en ADPS0 kan je de deelfactor van de ADC prescaler instellen
  - Door deze in te stellen als 011 wordt een deelfactor van 8 bekomen. Hiermee wordt de klok van de ADC ingesteld met klokfrequentie 2 MHz (16 MHZ / 8)

- Met het AMUX-register kan je volgende zaken instellen:
  - De reference voltage VREF
  - Left adjustment van resultaten (gebruik van 8-bit resultaten inpv 10-bit)
  - Het selecteren van het ingangskanaal

bit	7	6	5	4	3	2	1	0
ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0

  - MUX3, MUX2, MUX1, MUX0 instellen als 0000 => selectie multiplexkanaal 0
  - REFS1 en REFS0 worden gebruikt voor het instellen van VREF

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVcc with external capacitor on AREF pin
1	0	Reserved
1	1	Internal 1.1V (ATmega168/328) or 2.56V on (ATmega8)

Alternatief (maar minder snel) voor de instelling : analogReference(DEFAULT);

  - ADLAR (ADC Left Adjust Result)

Deze bit heeft invloed hoe de ADC-conversie wordt weergegeven als resultaat in het ADC dataregister. Door deze bit op 1 te plaatsen wordt deze conversie links uitgericht (waarde wordt omgezet naar 8-bit en in ADCH geplaatst)

```
18 void setup() {
19   pinMode(pwm0UTPin, OUTPUT); //PWM-pen 11 als uitgang
20
21   //instellen adcPrescaler met deelfactor 8 (011)
22   cbi(ADCSRA, ADPS2);
23   sbi(ADCSRA, ADPS1);
24   sbi(ADCSRA, ADPS0);
25
26   //ADC Left Adjust Result (ADLAR) deze bit op 1
27   //resultaat conversie bewaard in ADCH en ADCL en links uitgelijnd
28   sbi(ADMUX, ADLAR);
29
30   //reference selection bits 01
31   // (AVCC met externe condensator aan de AREF-pin gebruikt als VREF)
32   sbi(ADMUX, REFS0);
33   cbi(ADMUX, REFS1);
34
35   // instellen multiplexer ADC0
36   cbi(ADMUX, MUX0);
37   cbi(ADMUX, MUX1);
38   cbi(ADMUX, MUX2);
39   cbi(ADMUX, MUX3);
40
```

4-7 Genereren van nagalmeffect

- Fast PWM voor Timer2

TCCR2A
Timer/Counter1 Control Register A

COM2A1	COM2A0	COM2B1	COM2B0			WGM21	WGM20

TCCR2B
Timer/Counter Control Register B

FOC2A	FOC2B			WGM22	CS22	CS21	CS20

CS22	CS21	CS20	Clock Select
0	0	0	Timer/Counter stopped
0	0	1	clk_{T2S}
0	1	0	clk_{T2S} / 8
0	1	1	clk_{T2S} / 32
1	0	0	clk_{T2S} / 64
1	0	1	clk_{T2S} / 128
1	1	0	clk_{T2S} / 256
1	1	1	clk_{T2S} / 1024

COM2A1	COM2A0	Fast PWM Mode (WGM modes 3 and 7)
0	0	Normal Port Operation, OC2A disconnected
0	1	Mode 7: Toggle OC2A on Compare Match All other modes: Normal Port Operation, OC2A disconnected
1	0	Clear OC2A on Compare Match, set OC2A at BOTTOM (non-inverting mode)
1	1	Set OC2A on Compare Match, clear OC2A at BOTTOM (inverting mode)

Waveform Generation Mode

Mode	WGM22	WGM21	WGM20	Timer / Counter Mode of Operation	TOP	Update of OCR2A at	TOV Flag Set on
0	0	0	0	Normal	0xFF	Immediate	0xFF
1	0	0	1	PWM, Phase Correct	0xFF	TOP	0x00
2	0	1	0	CTC	OCR2A	Immediate	0xFF
3	0	1	1	Fast PWM	0xFF	BOTTOM	0xFF
4	1	0	0	Reserved			
5	1	0	1	PWM, Phase Correct	OCR2A	TOP	0x00
6	1	1	0	Reserved			
7	1	1	1	Fast PWM	OCR2A	BOTTOM	TOP

```
41  // fast PWM voor Timer2
42  cbi (TCCR2A, COM2A0);
43  sbi (TCCR2A, COM2A1);
44
45  sbi (TCCR2A, WGM20);
46  sbi (TCCR2A, WGM21);
47  cbi (TCCR2B, WGM22);
48
49  //instellen prescaler Timer2
50  sbi (TCCR2B, CS20);
51  cbi (TCCR2B, CS21);
52  cbi (TCCR2B, CS22);
53
54  // activeren PWM-port Timer2
55  sbi (DDRB, 3); //digitale pen 11 als uitgang
```

- Activeren PWM-poort voor Timer 2
  - POORTB bevat de digital pins 8 tot 13
  - sbi (DDRB, 3) plaatst digitale pin 11 (3de positie in poortregister) als OUTPUT

4-7 Genereren van nagalmeffect

- Uitschakelen Timer0-interrupt om vertragingen te vermijden
  - TIMSK0 controleert welke interrupts zijn enabeld.

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

Timer/Counter Interrupt Mask Register

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2

Timer/Counter Interrupt Mask Register 2

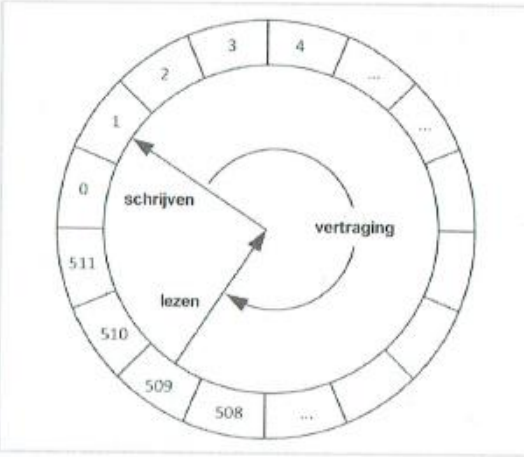
- valAnalog = analogSample;
  - analogSample bevat de geconverteerde samplewaarde van het analoog signaal dat gestockeerd werd in ADCH

```
57  //cli(); //interrupts uit om vervorming te voorkomen
58  cbi (TIMSK0, TOIE0); // Timer0 uit (vertraging vermijden)
59  sbi (TIMSK2, TOIE2); // activeren Timer2 interrupt
60
61  valAnalog = analogSample;
```

4-7 Genereren van nagalmeffect

- Eerst wordt er gewacht tot er een sample is geconverteerd, zodra deze beschikbaar is wordt verdergegaan in de loop; eerst wordt adcSample terug op false geplaatst
- Schaal vertraging van de sample vooraleer deze wordt uitgelezen
  - Aan analog input 2 wordt een potentiometer aangesloten die verbonden is met de 5 V.
  - De geconverteerde waarde van deze potmeter ligt tussen 0 en 255 (1 byte)
  - De const byte halfMax heeft als waarde 127
  - Stel dat de ringbuffer op positie 1 staat en hierin de byte met waarde 120 is gestockeerd. Stel dat potSample als waarde 200 heeft en valAnalog als waarde 180
  - valPot = (127 – 120) * 200/255 = 5,49 => 5 vermits valPot is een integer
  - valAnalog = 180 + 5 = 185
  - Controle of valAnalog groter is dan +127 (halfMax) of kleiner dan -127; in dit geval is het groter dan 127 => valAnalog begrenst tot waarde 127
  - Offset optellen bij PWM-buffer
    - Pwm_BUFFER bevat de waarde die in het OCR2A-register geplaatst wordt om fast-PWM te creëren
    - pwmBuffer = 127 + 127 = 254
    - Deze waarde wordt op positie 1 (voorbeeldwaarde) van de ringbuffer geplaatst.

```
65  void loop() {
66    while (!adcSample) {}
67    adcSample = false;
68
69    // schaal vertraagd sample met potmeterwaarde
70    valPot = (halfMax - ringBuff[indexCounter]) * potSample / 255;
71    valAnalog = halfMax - analogSample;
72    valAnalog = valAnalog + valPot;
73
74    // begrenzen analoge waarde (min/max)
75    if (valAnalog < -halfMax) valAnalog = -halfMax;
76    if (valAnalog > halfMax) valAnalog = halfMax;
77
78    // offset optellen bij PWM-buffer
79    pwmBuf = halfMax + valAnalog;
80    ringBuff[indexCounter] = pwmBuf;
81    indexCounter++;
```



4-7 Genereren van nagalmeffect

- Beperken bufferindex tot interval 0-511
  - 511 = 1 1111 1111 => door de and-functie zal bij overflow begint de teller terug vanaf 0 beginnen te tellen
- OCR2A krijgt de waarde van pwmBuff (254 in vb) Deze wordt ook opgeslagen in de ringbuffer en nadat de ring doorlopen is wordt deze waarde terug gebruikt om de sample die dan aanwezig is van echo te voorzien.
- Met valPot kan je de sterkte van het galmeffect instellen (luider echo of minder luid)

```
83    // beperken bufferindex tot het interval 0 - 511
84    indexCounter = indexCounter & 511;
85
86    OCR2A = pwmBuf; //schrijf huidige sample naar PWM-kanaal
87
88  }
```

4-7 Genereren van nagalmeffect

- Opstarten interrupt service routine bij Timer2 overflow vector
  - PORTB = PORTB | 1
  - Maakt gebruik van secundaire functie PB0

```
90  //ophalen analoge waarde en potmeterwaarde
91  ISR(TIMER2_OVF_vect) {
92    PORTB = PORTB | 1;
93    div2 = !div2; //reduceer samplefrequentie met factor 2
94    if (div2) {
95      selectChannel = !selectChannel;
96      if (selectChannel) {
97        potSample = ADCH;
98        sbi (ADMUX, MUX0); // stel multiplexer in op ADC0
99        }
100       else {
101         analogSample = ADCH;
102         cbi (ADMUX, MUX0); // multiplexer instellen op ADC0
103         adcSample = true;
104       }
105       sbi (ADCSRA, ADSC); //start volgende omzetting
106       }
107    }
```

MUX2	MUX1	MUX0	Selected Input
0	0	0	ADC0
0	0	1	ADC1
0	1	0	ADC2
0	1	1	ADC3
1	0	0	ADC4
1	0	1	ADC5
1	1	0	ADC6
1	1	1	ADC7

4-7 Genereren van nagalmeffect

- ADCSRA = ADC Control en Status Register A
  - Met ADPS2, ADPS1 en ADPS0 kan je de deelfactor van de ADC prescaler instellen
  - Door deze in te stellen als 011 wordt een deelfactor van 8 bekomen. Hiermee wordt de klok van de ADC ingesteld met klokfrequentie 2 MHz (16 MHZ / 8)

- Met het AMUX-register kan je volgende zaken instellen:
  - De reference voltage VREF
  - Left adjustment van resultaten (gebruik van 8-bit resultaten inpv 10-bit)
  - Het selecteren van het ingangskanaal

bit	7	6	5	4	3	2	1	0
ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0

  - MUX3, MUX2, MUX1, MUX0 instellen als 0000 => selectie multiplexkanaal 0
  - REFS1 en REFS0 worden gebruikt voor het instellen van VREF

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVcc with external capacitor on AREF pin
1	0	Reserved
1	1	Internal 1.1V (ATmega168/328) or 2.56V on (ATmega8)

Alternatief (maar minder snel) voor de instelling : analogReference(DEFAULT);

  - ADLAR (ADC Left Adjust Result)

Deze bit heeft invloed hoe de ADC-conversie wordt weergegeven als resultaat in het ADC dataregister. Door deze bit op 1 te plaatsen wordt deze conversie links uitgericht (waarde wordt omgezet naar 8-bit en in ADCH geplaatst)

```
18 void setup() {
19   pinMode(pwm0UTPin, OUTPUT); //PWM-pen 11 als uitgang
20
21   //instellen adcPrescaler met deelfactor 8 (011)
22   cbi(ADCSRA, ADPS2);
23   sbi(ADCSRA, ADPS1);
24   sbi(ADCSRA, ADPS0);
25
26   //ADC Left Adjust Result (ADLAR) deze bit op 1
27   //resultaat conversie bewaard in ADCH en ADCL en links uitgelijnd
28   sbi(ADMUX, ADLAR);
29
30   //reference selection bits 01
31   // (AVCC met externe condensator aan de AREF-pin gebruikt als VREF)
32   sbi(ADMUX, REFS0);
33   cbi(ADMUX, REFS1);
34
35   // instellen multiplexer ADC0
36   cbi(ADMUX, MUX0);
37   cbi(ADMUX, MUX1);
38   cbi(ADMUX, MUX2);
39   cbi(ADMUX, MUX3);
40
```

4-7 Genereren van nagalmeffect

- Fast PWM voor Timer2

TCCR2A
Timer/Counter1 Control Register A

COM2A1	COM2A0	COM2B1	COM2B0			WGM21	WGM20

TCCR2B
Timer/Counter Control Register B

FOC2A	FOC2B			WGM22	CS22	CS21	CS20

CS22	CS21	CS20	Clock Select
0	0	0	Timer/Counter stopped
0	0	1	clk_{T2S}
0	1	0	clk_{T2S} / 8
0	1	1	clk_{T2S} / 32
1	0	0	clk_{T2S} / 64
1	0	1	clk_{T2S} / 128
1	1	0	clk_{T2S} / 256
1	1	1	clk_{T2S} / 1024

COM2A1	COM2A0	Fast PWM Mode (WGM modes 3 and 7)
0	0	Normal Port Operation, OC2A disconnected
0	1	Mode 7: Toggle OC2A on Compare Match All other modes: Normal Port Operation, OC2A disconnected
1	0	Clear OC2A on Compare Match, set OC2A at BOTTOM (non-inverting mode)
1	1	Set OC2A on Compare Match, clear OC2A at BOTTOM (inverting mode)

Waveform Generation Mode

Mode	WGM22	WGM21	WGM20	Timer / Counter Mode of Operation	TOP	Update of OCR2A at	TOV Flag Set on
0	0	0	0	Normal	0xFF	Immediate	0xFF
1	0	0	1	PWM, Phase Correct	0xFF	TOP	0x00
2	0	1	0	CTC	OCR2A	Immediate	0xFF
3	0	1	1	Fast PWM	0xFF	BOTTOM	0xFF
4	1	0	0	Reserved			
5	1	0	1	PWM, Phase Correct	OCR2A	TOP	0x00
6	1	1	0	Reserved			
7	1	1	1	Fast PWM	OCR2A	BOTTOM	TOP

```
41  // fast PWM voor Timer2
42  cbi (TCCR2A, COM2A0);
43  sbi (TCCR2A, COM2A1);
44
45  sbi (TCCR2A, WGM20);
46  sbi (TCCR2A, WGM21);
47  cbi (TCCR2B, WGM22);
48
49  //instellen prescaler Timer2
50  sbi (TCCR2B, CS20);
51  cbi (TCCR2B, CS21);
52  cbi (TCCR2B, CS22);
53
54  // activeren PWM-port Timer2
55  sbi (DDRB, 3); //digitale pen 11 als uitgang
```

- Activeren PWM-poort voor Timer 2
  - POORTB bevat de digital pins 8 tot 13
  - sbi (DDRB, 3) plaatst digitale pin 11 (3de positie in poortregister) als OUTPUT

4-7 Genereren van nagalmeffect

- Uitschakelen Timer0-interrupt om vertragingen te vermijden
  - TIMSK0 controleert welke interrupts zijn enabeld.

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
TIMSK0	-	-	-	-	-	OCIE0B	OCIE0A	TOIE0

Timer/Counter Interrupt Mask Register

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
TIMSK2	-	-	-	-	-	OCIE2B	OCIE2A	TOIE2

Timer/Counter Interrupt Mask Register 2

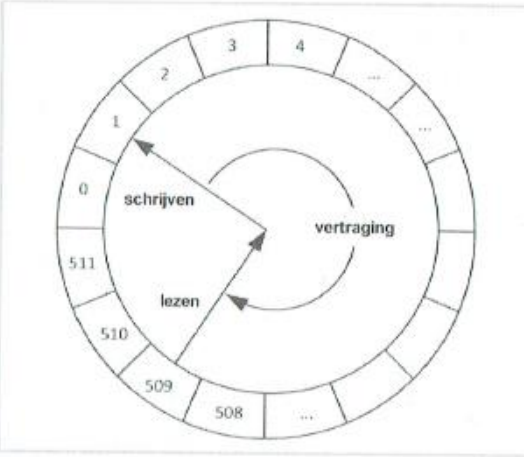
```
57  //cli(); //interrupts uit om vervorming te voorkomen
58  cbi (TIMSK0, TOIE0); // Timer0 uit (vertraging vermijden)
59  sbi (TIMSK2, TOIE2); // activeren Timer2 interrupt
60
61  valAnalog = analogSample;
```

- valAnalog = analogSample;
  - analogSample bevat de geconverteerde samplewaarde van het analoog signaal dat gestockeerd werd in ADCH

4-7 Genereren van nagalmeffect

- Eerst wordt er gewacht tot er een sample is geconverteerd, zodra deze beschikbaar is wordt verdergegaan in de loop; eerst wordt adcSample terug op false geplaatst
- Schaal vertraging van de sample vooraleer deze wordt uitgelezen
  - Aan analog input 2 wordt een potentiometer aangesloten die verbonden is met de 5 V.
  - De geconverteerde waarde van deze potmeter ligt tussen 0 en 255 (1 byte)
  - De const byte halfMax heeft als waarde 127
  - Stel dat de ringbuffer op positie 1 staat en hierin de byte met waarde 120 is gestockeerd. Stel dat potSample als waarde 200 heeft en valAnalog als waarde 180
  - valPot = (127 – 120) * 200/255 = 5,49 => 5 vermits valPot is een integer
  - valAnalog = 180 + 5 = 185
  - Controle of valAnalog groter is dan +127 (halfMax) of kleiner dan -127; in dit geval is het groter dan 127 => valAnalog begrenst tot waarde 127
  - Offset optellen bij PWM-buffer
    - Pwm_BUFFER bevat de waarde die in het OCR2A-register geplaatst wordt om fast-PWM te creëren
    - pwmBuffer = 127 + 127 = 254
    - Deze waarde wordt op positie 1 (voorbeeldwaarde) van de ringbuffer geplaatst.

```
65  void loop() {
66    while (!adcSample) {}
67    adcSample = false;
68
69    // schaal vertraagd sample met potmeterwaarde
70    valPot = (halfMax - ringBuff[indexCounter]) * potSample / 255;
71    valAnalog = halfMax - analogSample;
72    valAnalog = valAnalog + valPot;
73
74    // begrenzen analoge waarde (min/max)
75    if (valAnalog < -halfMax) valAnalog = -halfMax;
76    if (valAnalog > halfMax) valAnalog = halfMax;
77
78    // offset optellen bij PWM-buffer
79    pwmBuf = halfMax + valAnalog;
80    ringBuff[indexCounter] = pwmBuf;
81    indexCounter++;
```



4-7 Genereren van nagalmeffect

- Beperken bufferindex tot interval 0-511
  - 511 = 1 1111 1111 => door de and-functie zal bij overflow begint de teller terug vanaf 0 beginnen te tellen
- OCR2A krijgt de waarde van pwmBuff (254 in vb) Deze wordt ook opgeslagen in de ringbuffer en nadat de ring doorlopen is wordt deze waarde terug gebruikt om de sample die dan aanwezig is van echo te voorzien.
- Met valPot kan je de sterkte van het galmeffect instellen (luider echo of minder luid)

```
83    // beperken bufferindex tot het interval 0 - 511
84    indexCounter = indexCounter & 511;
85
86    OCR2A = pwmBuf; //schrijf huidige sample naar PWM-kanaal
87
88  }
```

4-7 Genereren van nagalmeffect

- Opstarten interrupt service routine bij Timer2 overflow vector
  - PORTB = PORTB | 1
  - Maakt gebruik van secundaire functie PB0

```
90  //ophalen analoge waarde en potmeterwaarde
91  ISR(TIMER2_OVF_vect) {
92    PORTB = PORTB | 1;
93    div2 = !div2; //reduceer samplefrequentie met factor 2
94    if (div2) {
95      selectChannel = !selectChannel;
96      if (selectChannel) {
97        potSample = ADCH;
98        sbi (ADMUX, MUX0); // stel multiplexer in op ADC0
99        }
100       else {
101         analogSample = ADCH;
102         cbi (ADMUX, MUX0); // multiplexer instellen op ADC0
103         adcSample = true;
104       }
105       sbi (ADCSRA, ADSC); //start volgende omzetting
106       }
107    }
```

MUX2	MUX1	MUX0	Selected Input
0	0	0	ADC0
0	0	1	ADC1
0	1	0	ADC2
0	1	1	ADC3
1	0	0	ADC4
1	0	1	ADC5
1	1	0	ADC6
1	1	1	ADC7

Opgave 6:

Analyseer het programma om galm op te wekken.

a) Geef aan (aan de hand van de code) waar de potmeter aangesloten moet worden om galmdiepte te bepalen en waar de ingangsschakeling aangesloten moet worden om het signaal waar galm wordt toegevoegd in arduino te nemen.

b) Sluit het fast PWM-signaal aan een versterker die opgebouwd is rondom een LM386

c) Test de schakeling eerst door een toon op te wekken via een functiegenerator.

d) Pas de schakeling aan zodat je een microfoon kunt gebruiken als audiobron

e) Kies een sample (waf-file) en speel deze via arduino af met galm en zonder galm

```
// echo produceren met arduino

//definieren macro's
#define cbi(PORT, bit) (PORT &= ~(1 << bit))
#define sbi(PORT, bit) (PORT |= 1 << bit)

const int pwmOUTPin = 11; // PWM-uitgang
//const float pi = 3.1415926535;
const byte halfMax = 127;
boolean div2, selectChannel; // stuurvariabele voor sampling

// globaal toegankelijke interruptvariabelen
volatile boolean adcSample;
volatile byte potSample, analogSample;
int indexCounter, valAnalog, valPot;
byte pwmBuf, ringBuff[512]; //audio ringbuffer-array 8-bit

void setup() {
  pinMode(pwmOUTPin, OUTPUT); //PWM-pen 11 als uitgang

  //instellen adcPrescaler met deelfactor 8 (011)
  cbi(ADCSRA, ADPS2);
  sbi(ADCSRA, ADPS1);
  sbi(ADCSRA, ADPS0);
```

```
//ADC Left Adjust Result (ADLAR) deze bit op 1
//resultaat conversie bewaard in ADCH en ADCL en links uitgelijnd
sbi(ADMUX, ADLAR);

//reference selection bits 01
// (AVCC met externe condensator aan de AREF-pin gebruikt als VREF)
sbi(ADMUX, REFS0);
cbi(ADMUX, REFS1);

// instellen multiplexer ADC0
cbi(ADMUX, MUX0);
cbi(ADMUX, MUX1);
cbi(ADMUX, MUX2);
cbi(ADMUX, MUX3);

// fast PWM voor Timer2
cbi (TCCR2A, COM2A0);
sbi (TCCR2A, COM2A1);

sbi (TCCR2A, WGM20);
sbi (TCCR2A, WGM21);
cbi (TCCR2B, WGM22);

  //instellen prescaler Timer2
  sbi (TCCR2B, CS20);
  cbi (TCCR2B, CS21);
  cbi (TCCR2B, CS22);

  // activeren PWM-port Timer2
  sbi (DDRB, 3); //digitale pen 11 als uitgang

  //cli(); //interrupts uit om vervorming te voorkomen
  cbi (TIMSK0, TOIE0); // Timer0 uit (vertraging vermijden)
  sbi (TIMSK2, TOIE2); // activeren Timer2 interrupt

  valAnalog = analogSample;

}
```

```
82
83    // beperken bufferindex tot het interval 0 - 511
84    indexCounter = indexCounter & 511;
85
86    OCR2A = pwmBuf; //schrijf huidige sample naar PWM-kanaal
87
88  }
89
90  //ophalen analoge waarde en potmeterwaarde
91  ISR(TIMER2_OVF_vect) {
92    PORTB = PORTB | 1;
93    div2 = !div2; //reduceer samplefrequentie met factor 2
94    if (div2) {
95      selectChannel = !selectChannel;
96      if (selectChannel) {
97        potSample = ADCH;
98        sbi (ADMUX, MUX0); // stel multiplexer in op ADC0
99        }
100       else {
101         analogSample = ADCH;
102         cbi (ADMUX, MUX0); // multiplexer instellen op ADC0
103         adcSample = true;
104       }
105       sbi (ADCSRA, ADSC); //start volgende omzetting
106       }
107    }
```