

## Goed programmeren in C Deel 2, programma bouwen.

Een programma schrijf je niet, dat bouw je. Ik ben in mijn tot nu toe korte carrière bijzonder gefocust geweest op het voorkomen van bugs, op het ordenen van grote programma's en op het in kaart brengen van complexe processen.

In dit deel leer ik je hoe je op een zeer geordende wijze complexe processen kan coderen. Met een verkleinde kans op bugs. Bugs kunnen we nooit helemaal voorkomen, maar wel gedeeltelijk. We zijn immers altijd vatbaar voor typfouten en denkfouten. Ik help je ook met opsporen van bugs. Dat wat niet kunnen voorkomen, moeten we kunnen opsporen.

Ik beschrijf een 'bug' als een onverwachts bijeffect van een programma met negatieve gevolgen. Bugs kunnen grote bugs zijn (je programma start niet eens op) of bugs kunnen kleine bugs zijn (een kleine actie loopt niet precies zoals je wilt).

Soms zijn het deze kleine bugjes die kunnen leiden tot catastrofale gevolgen. Zo kan je trein zomaar van je pendelbaan afrijden en op de grond eindigen omdat je 'diodefuij' code niet helemaal deed wat het moest doen.

In dit deel leg ik ook het gebruik van macro's uit. Ik noem alle voordelen en nadelen en waarom ze zo ontzettend handig kunnen zijn. Ze kunnen namelijk voorkomen dat we dingen vergeten te tikken, ze kunnen tot bepaalde mate code voor ons 'genereren' die wij dan niet meer fout kunnen doen en ze kunnen zaken verbergen die niet relevant zijn.

Bugs kan je namelijk gedeeltelijk voorkomen door in te grijpen in je programmastructuur, door code te verdelen in kleinere behapbare blokjes. Een foutje valt makkelijker op in een klein blokjes dan in grote lap code. We verdelen grote stapjes onder in meerdere kleinere stapje die elke een simpele logica en een duidelijke syntax hebben. Daarbij bouwen we ook een structuur in die ons de controle geeft over welke blokjes code wanneer uitgevoerd moeten worden.

Bij programmeren is de 20-80 regel vaak toepasselijk. In praktijk is gebleken dat menig programmeurs slechts 20% van de tijd bezig zijn met het maken van nieuwe code en 80% van de tijd zijn ze bezig om hun bugs er uit te vissen en te testen.

Ik maak in mijn software geen gebruik meer van `millis()`. Ik heb een alternatief methode bedacht om gebruik te maken van software timers. Hoe dit precies werkt, staat uitgelegd in de bijlage. Je kan ten alle tijden gebruik blijven maken van `millis()` als je dat prettiger vindt.

# Inhoud

Goed programmeren in C Deel 2, programma bouwen .....	1
Hoofdstuk 1 State-machines .....	3
§1.1 state diagram opzetten .....	4
§1.2 De standaard implementatie .....	5
§1.3 ordening door functies .....	10
§1.4 Nog meer ordening, states retourneren status .....	14
§1.5 sub-states .....	16
§1.6 inter-state delays .....	19
§1.7 De default en idle state .....	20
Hoofdstuk 2, Macro's en mythes .....	22
§2.1 macro valkuilen .....	23
§2.2 Verschil tussen #define en const int .....	25
§2.3 Waarom zou je wel macro's willen gebruiken? .....	26
§2.4 De macro operators .....	31
§2.4.1 De enkele # macro operator .....	31
§2.4.2 De dubbele ## macro operator .....	31
§2.5 De state-machine macro's .....	32
§2.5.1 De stateFunction macro .....	32
§2.5.2 De entryState macro .....	32
§2.5.3 De onState macro: .....	32
§2.5.4 De exitState macro: .....	33
§2.5.5 STATE_MACHINE_BEGIN en STATE_MACHINE_END .....	34
§2.5.6 De State macro .....	36
Hoofdstuk 3 Een programma opbouwen .....	37
§3.1 state-diagram .....	37
§3.2 #include .....	37
§3.3 de Macro's .....	38
§3.4 de States (contantes) .....	38
§3.5 de variabelen .....	38
§3.6 de Functies .....	39
§3.6 De state-functies .....	39
§3.7 de state-machine .....	40
§3.8 void setup() .....	40
§3.9 void loop() .....	41
§3.10 timers.cpp & timer.h .....	41
§3.11 Eerste tussentijdse test .....	42
§3.12 state-functies invullen .....	44
§3.13 Herhaalbaarheid .....	46
Hoofdstuk 4 multi-taken .....	47
§4.1 code opsplitsen in meerdere bestanden .....	47
§4.1.1 static & extern .....	47
§4.1.2 source en header files .....	49
§4.1.4 source en header files in combinatie met classes .....	50
§4.1.5 #include .....	50
§4.2 state-machines opsplitsen in source en header files .....	52
§4.3 State-machines parallel uitvoeren .....	54
§4.3.1 Het principe .....	54
§4.3.2 communicatie tussen state-machines .....	57
§5 slotwoord .....	59
Bijlage 1 De software timers .....	60

## Hoofdstuk 1 State-machines.

Een state-machine is een bepaalde code structuur waarmee je verschillende acties kan uitvoeren in een geordende methode. Een zo'n actie noemen we een state. Een state-machine kan een zo'n state tegelijk uitvoeren. Als die state dan klaar is, kan de state-machine een andere state uitvoeren. Er kunnen meerdere state-machines parallel aan elkaar lopen zonder dat ze elkaar beïnvloeden.

Weet dat niet elk proces geschikt is om in een state-machine te plaatsen. Voor simpelere taken is dat overbodig. Je kan state-machines gebruiken voor bijvoorbeeld deze voorbeelden:

- Een aki of een ahob
- Meerdere knipper lichtjes van autootjes tegelijk regelen
- Een pendeldienst
- Afhandeling van seriële communicatie
- Dag en nacht regeling met onweer geluiden en licht effecten
- Menu's voor lcd schermen voor bijvoorbeeld een handregelaar.

Niet trein-gerelateerd:

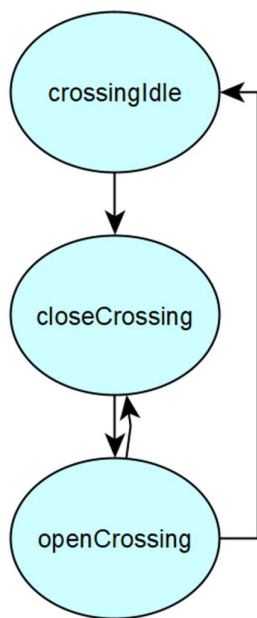
- Een digitale wekker
- Een zelfrijdende robot

### §1.1 state diagram opzetten.

Voordat je begint met het programmeren van een state-machines kan je een state-machine opzetten in de vorm van een simpele bollendiagram op papier of in een computerprogramma. Dit noem ik een 'state-diagram'.

In C is er meer dan een manier om een state-machine te bouwen. Een hiervan zijn om een switch-case te gebruiken en de ander is om function pointers te gebruiken. De function pointers laten we achterwege, ze zijn iets cryptischer en verder zijn ze niet echt bijzonder of beter dan wat ik zelf doe.

Voordat ik de state-machine maak in software doe ik eerst een state-diagram op de computer. Op het forum wilde iemand een overgang maken met een Arduino dus dat is wat we nu gaan doen. De overgang is opzich een interessant voorbeeld omdat we dingen parallel moeten gaan doen. Bij meer dan een spoor moeten we het inlezen van de detectors parallel gaan uitvoeren. We beginnen eerst met een state-diagram te maken voor een ahob. De detectors komen later aan bod.



We moeten twee dingen weten: wat doet elke state en wanneer springen we naar een andere state.

In de idle state doet de overgang helemaal niks behalve op treinen wachten. Deze state is actief zolang er geen treinen zijn. Vanuit idle state kunnen we alleen naar de closeCrossing state toe gaan.

In de closeCrossing state, moeten er lampen knipperen en de bomen moeten worden gesloten. Voor de bomen ga ik de servo library van Arduino gebruiken. Deze state blijft actief totdat alle treinen weer weg zijn. De closeCrossing state kan alleen maar naar de openCrossing state springen.

In de openCrossing state moeten de lampen blijven knipperen. De bomen moeten weer open worden gezet. Deze state is actief totdat de bomen open zijn, of totdat er een trein over het andere spoor binnen komt rijden. Deze state kan dus zowel naar de idle state als de closeCrossing state gaan (of 'transitioneren' om er een duur woord aan toe te kennen).

Het heeft mijn voorkeur om niet teveel informatie over de state flow (of flow conditie) bij de pijlen te zetten. Deze state diagrammen gebruiken we als leeswijzer om door code te navigeren. Deze moet daarom zo duidelijk als mogelijk zijn. Bovendien zullen we deze flow condities in code zetten met een zeer duidelijke syntax.

## §1.2 De standaard implementatie.

Voordat we beginnen met het coderen van de processen moeten we nadenken over alle IO. In dit voorbeeld hebben we de buitenste leds op de bomen, deze branden continu wanneer de bomen sluiten. De bomen en het kruis hebben ook elk twee leds die om en om wisselen. We hebben per spoor twee bezet melders en we hebben twee servo's. We hebben hier vier inputs, drie outputs en twee servo's nodig. Ik pak hiervoor willekeurige IO pinnen.

```
#define led1 10
#define blinkLed1 12
#define blinkLed2 13
#define detector1 2
#define detector2 3
#define detector3 4
#define detector4 5
```

En de pin modes instellen in de setup.

```
pinMode(led1, OUTPUT);
pinMode(blinkLed1, OUTPUT);
pinMode(blinkLed2, OUTPUT);
pinMode(detector1, INPUT_PULLUP);
pinMode(detector2, INPUT_PULLUP);
pinMode(detector3, INPUT_PULLUP);
pinMode(detector4, INPUT_PULLUP);
```

Zelf ben ik Marklinist en gebruik massadetectie voor de bezetmelders. Om dat te laten werken, heb ik de interne pull-up weerstanden nodig van de Arduino. En ik moet deze inputs geïnverteerd inlezen.

Voor de servo's moeten we het zelfde doen. We moeten ook de Servo library includen.

```
#include <Servo.h>
```

We moeten twee de servo pinnen definiëren.

```
#define servoPin1 6
#define servoPin2 7
```

We moeten twee servo objecten aanmaken.

```
Servo arm1;
Servo arm2;
```

En dan moeten we deze objecten nog 'attach'en aan de IO pinnen. We willen ook de begin positie van de servo's opgeven. En we gaan voor de servo's een variabele nodig hebben om de hoeken mee te regelen. Dit doen we ook in de void setup()

```
arm1.attach(servoPin1);
arm2.attach(servoPin2);

angle = beginAngle; // angle is een variabele, beginAngle een constante
arm1.write(angle);
arm2.write(angle);
```

Dit vind je ook in Rudy B's tutorial. Als laatste moeten we de variabele 'angle' en de constante 'beginAngle' maken.

```
#define beginAngle 0 // of net wat jou uitkomt  
byte angle;
```

We hebben nu alle IO geïnitieerd. Dus nu kunnen we beginnen met de states, de state variabele en de state-machine op te zetten. De states, of eigenlijk de namen van de states stoppen we in een enum. Je mag natuurlijk ook #defines of const int gebruiken als je daar gelukkiger van wordt. De 'state' variabele past in een byte en de switch-case krijgt drie cases, een voor elke state.

```
byte state;  
  
enum states {  
    crossingIdle,  
    closeCrossing,  
    openCrossing,  
};  
  
void ahob() {  
    switch(state) {  
        case crossingIdle:  
            // state code  
            break;  
  
        case closeCrossing:  
            // state code  
            break;  
  
        case openCrossing:  
            // state code  
            break;  
    }  
}
```

De functie ahob() is de state-machine. ahob() wordt continu aangeroepen vanuit void loop(). Afhankelijk van de waarde van 'state' wordt 1 van de 3 cases of states uitgevoerd.

De overgang heeft in totaal drie parallele processen. De twee processen voor de detectors moeten aan het ahob proces kunnen vertellen dat er een trein aankomt. Daar gaan we de volgende vlaggen voor gebruiken. Ik gebruik hiervoor bit fields.

```
struct {  
    unsigned int track1 : 1;  
    unsigned int track2 : 2;  
} occupied;  
//gebruik occupied.track1 = 1; of if(occupied.track1 == 1)
```

Deze vlaggen worden geset en ge'clear'ed door de twee andere processen. De ahob state-machine leest deze vlaggen alleen uit als input. Deze vlaggen worden dus gebruikt om meerdere state-machines met elkaar te laten communiceren.

Nu gaan we de eerste state in vullen.

```
case crossingIdle:  
    digitalWrite(led1, LOW);  
    digitalWrite(blinkLed1, LOW);  
    digitalWrite(blinkLed2, LOW);  
    if(occupied.track1 == 1 || occupied.track2 == 1) {  
        state = closeCrossing; }  
    break;
```

Deze state stuurt continu alle leds uit (om er zeker van te zijn dat ze uit staan) en houdt verder de bezet melder vlaggen in de gaten. Als of track1 bezet is of track2 bezet is, wordt de nieuwe state closeCrossing. Deze nieuwe state wordt dan als volgende uitgevoerd.

Nu gaan we de closeCrossing state maken.

```
case closeCrossing:
    digitalWrite(led1, HIGH); // continu lamp aan
    if(!blinkT) {
        blinkT = 50; // 0.5s
        if(digitalRead(blinkLed1)) {
            digitalWrite(blinkLed1, LOW);
            digitalWrite(blinkLed2, HIGH);
        }
        else {
            digitalWrite(blinkLed1, HIGH);
            digitalWrite(blinkLed2, LOW);
        }
    }
    if(!servoT) {
        servoT = 55; // elke 55ms
        if(angle < 90) angle ++;
        arm1.write(angle);
        arm2.write(angle);
    }
    if(occupied.track1 == 0 && occupied.track2 == 0) {
        state = openCrossing;
    }
    break;
```

We beginnen nu met de drie leds aan te sturen. Ik gebruik hiervoor wel mijn eigen decrementerende (aftellende) timer methode want met millis() blijven we op het toetsenboard rammen. Hoe dit precies werkt, wordt in §3.1 uitgelegd. Voor nu hoeft je alleen te weten dat de timers ten alle tijden aftellen naar 0 toe. Ik maak een blinkT voor de leds en een servoT voor de servo motors. blinkT krijgt een tijd basis van 100ms en servoT krijgt een tijdbasis van 1ms. Als je er gelukkiger van wordt mag je hier ook de millis functie gebruiken.

Ik laat de knipper lampen elke 0.5s toggelen. Ik weet niet precies hoelang de gemiddelde H0 ahob er over doet om te sluiten. Voor nu neem ik 5 seconde aan. In deze 5 secondes wil ik beide servo's 90 graden laten draaien. Dat betekent dat ik elke  $5000 / 90 = 55\text{ms}$  de servo's 1 graad moet verdraaien om een zo soepel als mogelijke beweging te simuleren.

De state mag naar openCrossing springen wanneer beide sporen leeg zijn.

Led1 wordt continu hoog gezet. blinkLed 1 en 2 worden elke 0.5s getoggeld. En de servo schuift elke 55ms 1 graad op tot een maximum van 90 graden. Pas wanneer beiden sporen vrij zijn, gaan de bomen weer open in de volgende state: openCrossing.

De openCrossing state ziet er bijna hetzelfde uit als de close state. Het voornaamste verschil is dat deze state zowel naar de idle als de vorige state kan springen.

De bomen kunnen ergens halverwege staan op het moment dat er een nieuwe trein komt. We willen dan niet dat ze opeens helemaal open of dicht springen. De globale variabele 'angle' die we hiervoor aangemaakt hebben, zorgt er voor dat dit niet gebeurt.

De openCrossing state staat op de volgende pagina.



```

case openCrossing:
    digitalWrite(led1, HIGH); // continu lamp aan

    if(!blinkT) {
        blinkT = 50; // 0.5s
        if(digitalRead(blinkLed1)) {
            digitalWrite(blinkLed1, LOW);
            digitalWrite(blinkLed2, HIGH);
        }
        else {
            digitalWrite(blinkLed1, HIGH);
            digitalWrite(blinkLed2, LOW);
        }
    }
    if(!servoT) {
        servoT = 55; // elke 55ms
        if(angle > 0) angle --;
        arm1.write(angle);
        arm2.write(angle);
    }
    if(angle == 0) { // als bomen open zijn..
        state = crossingIdle;
    }
    if(occupied.track1 || occupied.track2) { // als er een nieuwe trein
        state = closeCrossing; }           // komt voordat de bomen open
    break;                                // zijn..

```

Deze state gaat terug naar idle als de bomen stand 0 graden bereikt hebben (de angle is 0). Als echter voor die tijd een nieuwe trein binnen komt, gaan de bomen opnieuw dicht. In dat geval wordt de nieuwe state 'closeCrossing'.

### §1.3 ordening door functies

We hebben nu een werkende state-machine gemaakt om een ahob met servo's en leds aan te sturen. Je kan dit misschien mooi en handig vinden, maar de structuur kan nog vele malen mooier en netter. We hebben nu maar drie states, stel je je nu eens voor dat we er 50 hebben. Dan ga je een hele lange lap code krijgen waardoor moeilijk te navigeren is. De switch-case blijven we gebruiken maar ik ga nu functies toepassen om dit zoitje overzichtelijker te krijgen. Het eerste wat ik ga doen, is een functie maken voor leds te knipperen.

```
void blinkLeds(uint8_t blinkTime) {
    if(!blinkT) {
        blinkT = blinkTime; // 0.5s

        if(digitalRead(blinkLed1)) {
            digitalWrite(blinkLed1, LOW);
            digitalWrite(blinkLed2, HIGH);
        }
        else {
            digitalWrite(blinkLed1, HIGH);
            digitalWrite(blinkLed2, LOW);
        }
    }
}
```

De knipper tijd is nu een variabel argument geworden, genaamd 'blinkTime'. Dat heeft voor dit project misschien nog geen nut, maar het kan wel nut hebben voor andere projecten. Als we deze functie willen hergebruiken en een andere knipper tijd willen instellen, dan kan dat.

Dan maak ik een functie voor de servomotoren. Deze functie moet ik een argument meegeven om de servo's te laten sluiten of te laten openen. Dan kunnen we de functie gebruiken voor zowel het openen als het sluiten van de armen.

```
void controlArms(int8_t direction) {
    if(!servoT) {
        servoT = 55; // elke 55ms

        if(direction == 1) { // close
            if(angle < 90) angle++;
            arm1.write(angle);
            arm2.write(angle);
        }
        else {
            if(angle > 0) angle --;
            arm1.write(angle);
            arm2.write(angle);
        }
    }
}
```

Het argument 'direction' is er om te bepalen of de armen dicht of open moeten. '1' is dicht. Iets anders dan 1 is open.

Dan zet ik de functie calls in de states in plaats van de oude code en we krijgen:

```
void ahob(void) {
    switch(state) {
        case crossingIdle:
            digitalWrite(led1, LOW);
            digitalWrite(blinkLed1, LOW);
            digitalWrite(blinkLed2, LOW);
            if(occupied.track1 || occupied.track2) state = closeCrossing;
            break;

        case closeCrossing:
            digitalWrite(led1, HIGH); // continu lamp aan
            blinkLeds(50);
            controlArms(1);
            if(!occupied.track1 && !occupied.track2) state = openCrossing;
            break;

        case openCrossing:
            digitalWrite(led1, HIGH); // continu lamp aan
            blinkLeds(50);
            controlArms(-1);
            if(angle == 0)/*creatief uitlijnen mag */state = crossingIdle;
            if(occupied.track1 || occupied.track2) state = closeCrossing;
            break;
    }
}
```

Dit is al een stuk beter dan wat we eerst hadden. De functies blinkLeds() en controlArms() zouden we nu naar een ander bestand kunnen verhuizen. Hoe dat moet, wordt uitgelegd in §4.1.

Ik ben echter nog niet content met de huidige state-machine. Hij kan namelijk nog beter en nog overzichtelijker. Wat we nu nog steeds hebben, is dat states en state-machine samen 1 zijn. Nu gaan we met functies een scheiding maken tussen state-functie en state-machine. Dit doen we door de code tussen alle cases en breaks naar een functie te verplaatsen. Deze functie mag niet hetzelfde heten als de state zelf omdat de naam simpelweg bezet is. We mogen wel letters toevoegen aan de naam. Laat ik nu deze functies maken met “State” er achter geplakt, wat krijgen we dan?

```

void crossingIdleState(void) {
    digitalWrite(led1, LOW);
    digitalWrite(blinkLed1, LOW);
    digitalWrite(blinkLed2, LOW);
    if(occupied.track1 || occupied.track2) state = closeCrossing;
}

void closeCrossingState(void) {
    digitalWrite(led1, HIGH); // continu lamp aan
    blinkLeds(50);
    controlArms(1);
    if(!occupied.track1 && !occupied.track2) state = openCrossing;
}

void openCrossingState(void) {
    digitalWrite(led1, HIGH); // continu lamp aan
    blinkLeds(50);
    controlArms(-1);
    if(angle == 0) state = crossingIdle;
    if(occupied.track1 || occupied.track2) state = closeCrossing;
    break;
}

void ahob(void) {
    switch(state) {
        case crossingIdle:
            crossingIdleState();
            break;

        case closeCrossing:
            closeCrossingState();
            break;

        case openCrossing:
            openCrossingState();
            break;
    }
}

```

We zijn er nog niet. Ik ga een stap nemen die uitermate belangrijk is voor de ordening. Alle code waarin 'state' wordt veranderd, moet terug komen te staan in de switch-case. Waarom wordt duidelijk. Ik ga alle desbetreffende regels naar de switch-case verhuizen. Dat ziet er zo uit.

```

switch(state) {
case crossingIdle:
    crossingIdleState();
    if(occupied.track1 || occupied.track2) state = closeCrossing;
    break;

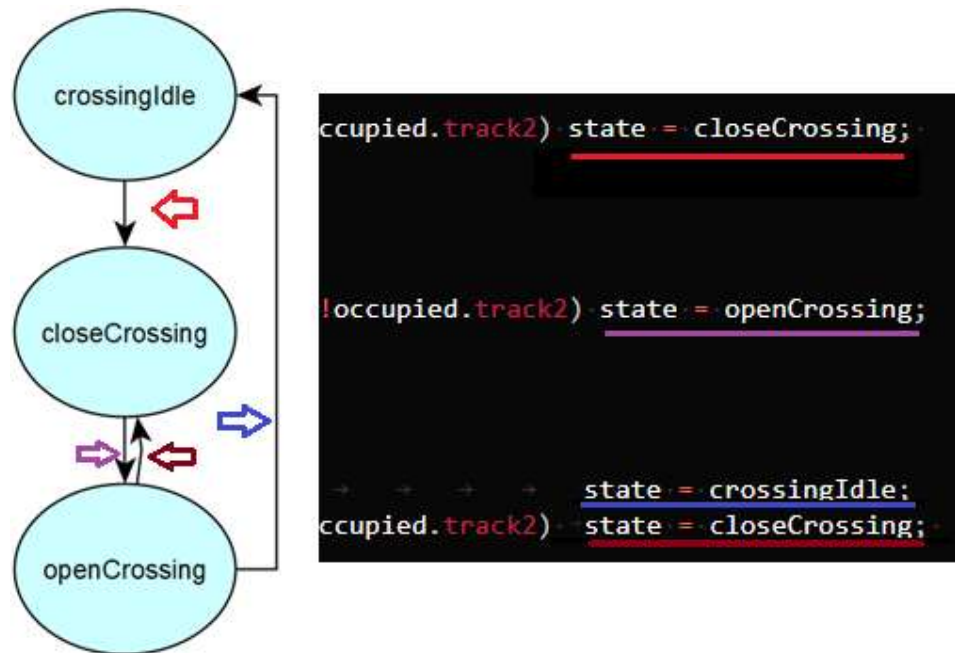
case closeCrossing:
    closeCrossingState();
    if(!occupied.track1 && !occupied.track2) state = openCrossing;
    break;

case openCrossing:
    openCrossingState();
    if(angle == 0) state = crossingIdle;
    if(occupied.track1 || occupied.track2) state = closeCrossing;
    break;
}

```

Wat je misschien opvalt is dat de lijnen met 'state = ...' nu 1 op 1 overeenkomt met de state diagram die we gemaakt hebben in §1.1

Nu zijn we goed op weg met een duidelijke state-machine. We kunnen nu ook makkelijker navigeren in de code. De bollen diagram die we eerder gemaakt hebben, komt nu 1 op 1 overeen met onze compacte switch-case.



Het is dit soort ordening die we nodig hebben. We kunnen nu makkelijk navigeren, we kunnen nu makkelijk wijzigingen aanbrengen, we kunnen makkelijk states toevoegen, veranderen of verwijderen. En gedurende deze acties hebben we naast ons een bollendiagram liggen om ons daarbij te helpen.

Wat nu ook erg fijn is dat de state-functie bijna hetzelfde heet als de state naam. De state 'openCrossing' heeft de bijbehorende state-functie 'openCrossingState'. Als ik nu in de switch-case openCrossing opzoek en selecteer, dan kan ik met de standaard zoek functie binnen 1 seconde springen naar de state-functie code. En ik kan net zo makkelijk weer terug springen naar de state-machine terug springen.

### §1.4 Nog meer ordening, states retourneren status

Er zijn echter nog meer kleine stapjes die ik wil maken. Voor de ahob state-machine, volstaat onze huidige constructie. Maar hij kan nog mooier. Waar we nu naar toe willen, is dat de state-functies een status retourneren. Een state-functie is klaar of hij is niet klaar. Daarvoor moeten we de state-functies een 'bool' laten retourneren. Daarvoor moeten we de 'void's' van de state-functies veranderen in 'bool's' en de state-functies moeten 'true' of 'false' retourneren.

```
bool crossingIdleState(void) {
    if(occupied.track1 || occupied.track2) return true;
    else return false;
}

bool closeCrossingState(void) {
    digitalWrite(led1, HIGH); // continu lamp aan
    blinkLeds(50);
    controlArms(1);
    if(!occupied.track1 && !occupied.track2) return true;
    else return false;
}

bool openCrossingState(void) {
    digitalWrite(led1, HIGH); // continu lamp aan
    blinkLeds(50);
    controlArms(-1);
    if(angle == 0 || occupied.track1 || occupied.track2) return true;
    else return false;
}
```

De state-machine moet nu pas een nieuwe state selecteren wanneer een van de state-functies 'true' retourneert. Dit doen we de door de state-functie aan te roepen vanuit een if-statement. Dat ziet er zo uit:

```
switch(state) {
case crossingIdle:
    if(crossingIdleState()) {
        state = closeCrossing; }
    break;

case closeCrossing:
    if(closeCrossingState()) {
        state = openCrossing; }
    break;

case openCrossing:
    if(openCrossingState()) {
        if(angle == 0) state = crossingIdle;
        else          state = closeCrossing; }
    break;
}
```

Alle regels die nu 'state' aanpassen staan in de bodies van de if-statements. Pas wanneer een state-functie 'true' retourneert, is de if-statement waar en wordt een nieuwe state geselecteerd. Het voordeel van de methode is dat er minder code in de switch-case hoeft te staan. Vooral voor de eerste en tweede state is het niet meer nodig om te zeggen waarom we naar die state springen.

We kunnen namelijk in deze states toch maar naar een andere state springen. Zodoende hoeft de switch-case alleen te weten dat de state-functie klaar is.

Voor de derde state 'openCrossing' ligt het iets anders. Deze state kan naar twee andere states springen. We moeten daarom de condities verschaffen 'waarom'. In dit geval heb ik voor de angle gekozen omdat dat het minste code is. In de state-functie staat nu dus deze regel code.

```
if(angle == 0 || occupied.track1 || occupied.track2) return true;
```

De state-functie vertelt nu dat hij klaar als een van deze drie condities waar is. Vervolgens kijkt de switch-case naar een van deze condities om te kijken welke state hij hierna moet kiezen met:

```
if(angle == 0) state = crossingIdle;  
else          state = closeCrossing;
```

Deze structuur is erg wenselijk voor een goede ordening. Nu staat de informatie daar waar we het willen hebben en waar we het nodig hebben. En nogmaals, voor zo'n klein project is het een kleine toevoeging maar wanneer je 20+ states heb, dan ga je dit erg graag willen. Bovendien, we kunnen nu namelijk nog iets extra's doen. Wat, dat leg ik uit in de volgende paragraaf.

## §1.5 sub-states

Op dit moment zijn er slechts twee inefficiënte eigenschappen te vinden in deze hele code. En het zijn maar kleine. De leds die op het uiteinde van de ahob zitten, die branden continu. Wat we eigenlijk 'fout' doen, is dat we deze continu aansturen. Vooral in de idle state sturen we drie leds continu uit, terwijl 1x zou volstaan. Bovendien kunnen we het ook nog zo bouwen dat het niet eens nodig is om de idle state dit te laten doen.

Voor veel state-machine applicaties is het wenselijk om een blokje code te hebben in de state-machine wat slechts 1 malig wordt uitgevoerd. Dit blokje code is wat ik noem de 'entryState'. Het blokje code wat continu wordt uitgevoerd, waarin ook blinkLed() en controlArms() staan, noem ik de 'onState'. En met de laatste wijziging van de vorige paragraaf hebben we stiekem de 'exitState' gemaakt. Dit blokje code wordt 1 malig uitgevoerd voordat de state klaar is en true retourneert.

```
if(angle == 0 || occupied.track1 || occupied.track2) return true;
```

Deze regel hebben we al gezien in een state-functie, het enige wat ik nu toe, is accolades toevoegen en een beetje commentaar.

```
if(angle == 0 || occupied.track1 || occupied.track2) {  
    // exit state code gaat hier  
  
    return true;  
}
```

Wat wij er in ons geval mee kunnen, is leds uitschakelen. Als we nu in de 'exitState' van openCrossingState alle leds uitzetten, hoeven we dit niet meer te doen in de idle state.

```
bool openCrossingState(void) {  
    blinkLeds(50);  
    controlArms(-1);  
    if(angle == 0 || occupied.track1 || occupied.track2) {  
        digitalWrite(led1, LOW); // zet alle lampen uit  
        digitalWrite(blinkLed1, LOW);  
        digitalWrite(blinkLed2, LOW);  
        return true;  
    }  
    else return false;  
}
```

Zoals je kan zien, zetten we alle drie de leds uit, voordat er 'true' wordt geretourneerd.

De crossingIdle state kan er dan zo uit zien:

```
bool crossingIdleState(void) {  
    if(occupied.track1 || occupied.track2) return true;  
    else return false;  
}
```

Het enige wat de idle state nu nog maar doet, is wachten tot er een trein komt. En doet verder niks overbodigs.

Nu moeten we alleen nog de 'entryState' realiseren. Om het nut hiervan goed te demonstreren, laten we elke state-functie een bericht op het scherm printen. Zonder de te maken entryState constructie, zou de state-functie continu een bericht blijven printen totdat de state wordt verlaten.



Om de entryState te maken hebben we een nieuwe vlag nodig. Ik noem deze 'runOnce' en ik initialiseer hem op 'true'.

```
bool runOnce = true;
```

Omdat de 'entryState' als eerste wordt uitgevoerd, bouwen we deze in aan het begin van de state-functies. Ik laat nu alleen de openCrossingState-functie zien die nu is voorzien van een 'entryState'.

```
bool openCrossingState(void) {
    if(runOnce) {
        runOnce = false;
        // entryState code
        digitalWrite(led1, HIGH); // zet de buitenste lampen aan
        Serial.println("led 1 wordt nu eenmalig aangezet");
    }
    // onState code
    blinkLeds(50);
    controlArms(-1);
    if(angle == 0 || occupied.track1 || occupied.track2) {
        // exitState code
        digitalWrite(led1, LOW); // zet alle lampen uit
        digitalWrite(blinkLed1, LOW);
        digitalWrite(blinkLed2, LOW);
        return true;
    }
    else return false;
}
```

De runOnce vlag is true op het moment dat er een nieuwe state wordt uitgevoerd (Dit bouwen we zo in). De if-statement wordt dan als eerste uitgevoerd. In de body zetten we als eerste deze 'runOnce' flag uit zodat deze body niet nog een keer wordt uitgevoerd. In de body zetten we nu led1 eenmalig aan en we printen dat op de seriële monitor.

Er is nog een klein detailtje dat we moeten doen. Wanneer een state klaar is, moet de runOnce vlag opnieuw worden geset voor de nieuwe state. Dit kunnen we doen, voordat we true retourneren in de exitState, maar stel nou dat je dit op een dag vergeet? Dan loop je tegen de lamp aan, want je hebt een bug veroorzaakt. De entryState van de volgende state-functie wordt niet uitgevoerd en led1 op de AHOB brandt niet meer.

Om te voorkomen dat we ooit deze bug creëren, schrijf ik een kleine functie, genaamd 'nextState()'. Deze functie ziet er zo uit:

```
void nextState(uint8_t newState) {
    state = newState;
    runOnce = true;
}
```

Deze nextState-functie gaan we nu in de switch-case of state-machine zetten. Om van state te wisselen.

```
switch(state) {
case crossingIdle:
    if(crossingIdleState()) {
        nextState(closeCrossing); }
    break;

case closeCrossing:
    if(closeCrossingState()) {
        nextState(openCrossing); }
    break;

case openCrossing:
    if(openCrossingState()) {
        if(angle == 0) nextState(crossingIdle);
        else           nextState(closeCrossing); }
    break;
}
```

Als we nu altijd in plaats van "state =" tikken, "nextState()" tikken, kunnen we niet meer vergeten om de 'runOnce' vlag te zetten. Deze functie doet dat voor ons. Deze bug is nu permanent voorkomen.

Op mijn werk hebben ik en mijn collega's ondervonden dat deze implementatie ontzettend efficiënt is. Alles wat we nu willen programmeren kunnen we zonder moeite doen. We hoeven op dit punt niet meer over de structuur na te denken, want deze structuur is nu af. Vooral de entryState is bijzonder onmisbaar gebleken. De entryState zorgt er ook dat alle acties in een state passen. Dit maakt onze states ook modulair.

Sommige mensen gebruiken geen entry state. Zij doen dan een handeling in een bepaalde state en wachten in een andere state af of er iets gebeurd. Daardoor zijn die twee states nu met elkaar 'getrouwd'. Samen doen ze 1 ding. Op werk moeten we vaak met een CAN-bus bericht een stappenmotor aansturen en dan moeten we continu in de gaten houden of de stapper motor in positie is. Als je het CAN-bus bericht continu zou verzenden, dan zou je verzend buffer kunnen overflowen en dat komt niet meer goed. De entry state stelt ons in staat om deze hele actie te beperken tot slechts 1 state.

Zonder de entry states is het bij wijzigingen van een state-machine vaak nodig om code blokken te knippen en te plakken. Hierbij is er een grotere kans dat je iets verkeerd doet of iets vergeet en zo een bug veroorzaakt.

## §1.6 inter-state delays

Wanneer we op werk geen stappen motors aan het aansturen zijn, zijn we bezig met het aansturen van pneumatische cilinders. Dan zetten we een output hoog waardoor uiteindelijk een cilinder een bepaalde stand zal bereiken. Dit kan eventjes duren. Wij weten nu onder tussende dat we een vertraging in een state kunnen inbouwen met onze aftellende timers.

Voor het gemak en de flexibiliteit had ik ook een methode bedacht om een bepaalde delay (niet blokkerend) te maken tussen twee opvolgende states. Voordat ik laat zien hoe dat werkt, laat ik eerst zien hoe we dat kunnen gebruiken. Ik had hiervoor een aanpassing gemaakt aan de nextState-functie. We kunnen er nu een 2<sup>e</sup> argument aan toe voegen met een bepaalde tijdsvertraging.

Als we onze state-machine uit de vorige paragraaf aanpassen met deze 'inter-state delay' dan ziet hij er zo uit:

```
case crossingIdle:
    if(crossingIdleState()) {
        nextState(closeCrossing, 10); }
    break;

case closeCrossing:
    if(closeCrossingState()) {
        nextState(openCrossing, 10); }
    break;

case openCrossing:
    if(openCrossingState()) {
        if(angle == 0) nextState(crossingIdle, 10);
        else          nextState(closeCrossing, 20); }
    break;
```

De enige toevoegingen die we zien, zijn de nieuwe getallen in nextState. Voor de ahob is dit niet echt nodig, maar op werk maken wij hier veelvuldig gebruik van om op die cilinders te wachten.

Dan nu de werking. De state-functies blijven zelf onveranderd. We veranderen hier alleen de state-machine zelf en we brengen een wijziging aan aan nextState(). Deze wijziging ziet er zo uit:

```
static void nextState(unsigned char _state, unsigned char _interval) {
    runOnce = true;
    if(_interval) {
        enabled = false;
        ahobT = _interval; }
    state = _state; }
```

We zien hier twee nieuwe variabelen. Een bool genaamd 'enabled' en een nieuwe software timer genaamd ahobT (hoofdletter T van timer). Als we nu een tijd vertraging opgeven (\_interval) dan zetten we deze enabled vlag op false en \_interval wordt hier in onze timer, ahobT, geladen.

Dan passen we de state-machine aan naar het volgende voorbeeld.

```
void ahob() {
  if(enabled == false) {
    if(ahobT == 0) enabled = true;
  }
  else {
    switch(state) {
      case crossingIdle:
        if(crossingIdleState()) {
          nextState(closeCrossing, 10); }
        break;
    }
  }
  ...
}
```

Je ziet hier dat de switch-case die onze state-machine vormt nu in een else body staat. Als we dus nextState uitvoeren met een bepaalde interval waarde dan wordt enabled false. Als enabled false is, wordt de state-machine niet meer uitgevoerd.

Als dat het geval is, dan houden we continu in de gaten of de gezette timer ahobT 0 bereikt heeft. Zodra deze timer 0 bereikt, wordt enabled weer op true geset en zal de state-machine, na de delay, een nieuwe state-functie gaan uitvoeren.

We kunnen nu dus met onze software timers een delay inbouwen in een state maar ook tussen de states. Het voordeel van deze laatste methode is dat we slechts een enkel getalletje moeten aanpassen.

Voor nu is het hoofdstuk over state-machines af. We gaan deze state-machine nog mooier en nog netter maken in §2.5 met macro's. Deze macro's zijn veel gecompliceerder dan een simpele

```
#define ledpin 13
```

### §1.7 De default en idle state

Stel nu dat iets verpest in je code en je 'state' variabele krijgt een waarde waarvoor er geen state bestaat. De switch-case zou dan niets meer uitvoeren totdat 'state' weer een waarde van een bestaande state krijgt. Als je goed programmeert en altijd consistent de woorden van constanten gebruikt, zou dit niet moeten kunnen voorkomen. Maar wat als? Het zou in ieder geval fijn zijn om te weten, dat een state-machine niet meer werkt. Dan weten we in ieder geval waarna we opzoek moeten gaan om de bug te vinden. Daarvoor kunnen we de default label gebruiken.

```
void ahob() {
  if(enabled == false) {
    if(ahobT == 0) enabled = true;
  }
  else {
    switch(state) {
      default:
        Serial.println("unknown state entered, state is IDLE now");
        state = crossingIdle;
      case crossingIdle:
```

In het geval dat 'state' dus een rare of 'ongedefinieerde' waarde krijgt, zal de default label worden uitgevoerd. De code achter deze label zet op het scherm dat er een 'unknown state' is geselecteerd en dat 'state' op IDLE is gezet. Omdat deze idle state er onder staat, hebben we geen break; nodig en kunnen we veilig doorvallen naar deze idle state. Wij zijn er dan op geattendeerd dat 'state' ooit een rare waarde heeft gekregen en doordat we de state op idle zetten, kan de state-machine zich zelf hervatten.

## Hoofdstuk 2, Macro's en mythes

Macro's zijn een uiterst nuttige tool waarmee je de meest geweldige dingen kan doen. Aan de andere kant kunnen ze ook je leven tot een hel maken. Ze schijnen niet type-safe te zijn, dit is gedeeltelijk waar.

Een macro is slechts een methode om tekst te substitueren door iets anders. Deze ken je onder tussen wel:

```
#define output 8
```

Overall waar je output tikt, wordt onderhuids '8' neergezet voordat het programma compileert.

Je kan met macro's veel complexere dingen maken. Arduino heeft onderhuids ook veel macro's gebruikt. De 'bitSet()' en de 'bitClear()' functies, zijn geen functies dit zijn macro's. Elders op het internet kom je deze wel eens tegen SET(x), CLR(x) en TGL(x). De eerste twee zijn hetzelfde als de bitSet en bitClear macro's van Arduino. Ze zien er zo uit.

```
#define SET(port, pin) ((port) |= (1 << (pin)))  
#define CLR(port, pin) ((port) &= ~(1 << (pin)))  
#define TGL(port, pin) ((port) ^= (1 << (pin)))
```

Als je de poorten van een Arduino direct wilt aansturen in plaats van digitalWrite() te gebruiken, gebruik je dit soort macro's om dat snel te doen. Bijvoorbeeld ledPin 13 op een UNO is eigenlijk pin 5 van PORTB. Als je deze pin wilt toggelen, kan je dat met een macro doen.

```
TGL(PORTB, 5);
```

Dit is ongeveer 40x sneller in uitvoer dan:

```
digitalWrite(13, !digitalRead(13));
```

En het doet hetzelfde.

Wat opvallend is bij de macro definities is dat zowel port als pin ook tussen extra haakjes staan. Tevens zijn de macro's met hoofdletters getikt. Dit is beiden niet nodig in dit voorbeeld om te doen. Maar toch doen we dit en met hele goede redenen.

## §2.1 macro valkuilen

Macro's komen met vele valkuilen (valkuilen). Ik laat er een paar zien. Stel dat we iets willen hebben om een getal te kwadrateren. Daarvoor maken we een macro.

```
#define SQR(x) ((x) * (x))
```

Als we deze macro willen gebruiken doen we bijvoorbeeld:

```
byte x = SQR(5);  
Serial.println(x); // prints 25
```

Nu is dit helemaal prima, maar nu doe ik iets anders.

```
byte x = 4;  
x = SQR(++x);  
Serial.println(x); // prints 30
```

Waarom wordt hier nu 30 geprint en geen 25?

Als we nu zelf de macro vervangen door zijn definitie wat krijgen we dan?

```
byte x = 4;  
x = ((++x) * (++x)); // eerste x is 5, 2e x is 6  
Serial.println(x); // prints 30
```

De ++ operator hoogt eerst x op naar 5 voordat de code de waarde gebruikt. Maar x wordt nog een keer geïncrmenteerd naar 6. Het resultaat is nu 5 x 6 en dat wordt opgeslagen in 'x'. Dit kunnen we simpel voorkomen door geen macro's te gebruiken voor rekenkundige acties maar functies.

```
uint16_t sqr(uint8_t x) {  
    return (x) * (x);  
}  
byte x = 4;  
x = sqr(++x);  
Serial.println(x); // prints 25
```

X wordt nu eerst opgehoogd naar 5, dan als argument meegeven aan de functie waar hij verder niet wordt veranderd. De functie retourneert x \* x hetgeen hier correct verloopt.

Nu leg ik uit waarom je altijd en alles in () haakjes moet stoppen bij macro's. Stel we hebben deze macro:

```
#define DIV(a,b) a / b
```

En we proberen:

```
DIV(25,3+2); // we willen 25 / 5 = 5 zien, maar we krijgen 10
```

Dan krijgen we eigenlijk dit:

```
25 / 3 + 2; // dit is 10
```

Dit los je dus op door de () haken te plaatsen:

```
#define DIV(a,b) (a) / (b)  
(25) / (3 + 2); // dit is wel 5
```

Maar eigenlijk wil je voor deze rekenkundige actie ook een functie gebruiken.

Een andere macro valkuil is dat je per ongeluk een naam verzint die al gebruikt wordt. Stel ik maak een macro genaamd 'begin'

```
#define begin(x) ((x) * 10)
```

Deze kan je veilig gebruiken totdat je de Seriële communicatie opzet.

```
Serial.begin(115200);
```

Als je de macro uitschrijft krijgen we deze mooie constructie.

```
Serial.((115200) * 10);
```

En dit is natuurlijk onzin wat niet eens kan compileren. Als je pech hebt, dan krijg je iets wat wel kan compileren. Dan gaat je code volkomen onverwachts iets anders doen dan dat je wilt. Stel dat we deze macro gaan gebruiken.

```
#define begin(x) print(x)
```

Dan wordt:

```
Serial.begin(115200);
```

Omgezet in:

```
Serial.print(115200);
```

En dit compileert erg goed. Het doet alleen iets compleet anders dan wat je er wilt. In plaats van dat de seriële bus wordt geïnitieerd, wordt '1', '1', '5', '2', '0' en '0' in de buffer gestopt. Omdat de bus nu niet geïnitieerd is, wordt er niks naar het scherm gestuurd. Je seriële communicatie werkt niet meer en je hebt niet door dat dit komt door je macro!!

Je hebt zojuist geleerd waarom macro's dus heel erg vaak met HOOFDLETTERS worden getikt. De regel is dat alleen macro's uit alleen HOOFDLETTERS mogen bestaan. Zo kan je dus nooit dit soort conflicten krijgen. Als deze macro nu BEGIN had geheten, was er niks aan het handje.



## §2.2 Verschil tussen #define en const int

Voor constanten kan je behalve een enum gebruik maken van const int en een #define. Maar wat is nu het verschil hier tussen? In de praktijk maakt het vrij weinig uit.

Beschouw beide voorbeelden. De programma's doen het zelfde maar de een heeft een const int en de ander een macro

<pre>1 #define constante 13 2 3 void setup() { 4   constante = 5; 5 } 6 7 void loop() { 8 9 }</pre>	<pre>1 const int constante = 13; 2 3 void setup() { 4   constante = 5; 5 } 6 7 void loop() { 8 9 }</pre>
<p>lvalue required as left operand of assignment</p> <p>C:\Users\SKNIPP-1.HM\AppData\Local\Temp\arduino_modified_sketch_485038\sketch_jan21a:4: error: lvalue required as left operand of assignment</p> <pre>constante = 5;</pre> <p>exit status 1</p> <p>lvalue required as left operand of assignment</p>	<p>assignment of read-only variable 'constante'</p> <p>C:\Users\SKNIPP-1.HM\AppData\Local\Temp\arduino_modified_sketch_85116\sketch_jan21a:4: error: assignment of read-only variable 'constante'</p> <pre>constante = 5;</pre> <p>exit status 1</p> <p>assignment of read-only variable 'constante'</p>

Beiden leveren een compiler fout op. Ze geven wel verschillende fouten. De #define fout is enigszins iets cryptischer dan de andere. Nu zijn compiler fouten zelden niet cryptisch. We weten in ieder geval zeker dat we in lijn 4 iets gedaan hebben wat fout is. De const int geeft hier wel een zeer duidelijke foutmelding: "error: assignment of read-only variable 'constante'". Dus in dat opzicht is het inderdaad beter om een const int te gebruiken. Maar in praktijk maakt het dus niet zo veel uit.

Andere compilers echter kunnen minder flexibel zijn met macro's er bestaan compilers die niet altijd alle fouten er uit kunnen halen. Tevens laten sommigen van deze compilers ook niet toe dat je een const int mocht gebruiken als een case-label.

### §2.3 Waarom zou je wel macro's willen gebruiken?

Om te beginnen, wil ik eerst nog het volgende broodje aap citeren. Deze is tegen mij gebruikt:

“By using macros, you make your code harder to read because you use non-standard code”.

Dit is simpelweg een theorie, niets meer niets minder. Het is geen bewezen feit. Het kan wel waar zijn maar dat hoeft dus niet perse altijd zo te zien. Je kan code wel degelijk beter leesbaar maken met macro's.

Ik kwam zelf ooit met mijn eigen theorie:

“Als je twee lijntjes identiek code herhaaldelijk gebruik, maak er dan een macro van”.

Ik had hier ook geen gelijk mee. En het volgende voorbeeld liet me dat in zien. Ik maak zelf geen gebruik van de millis() functie. En daar heb ik redenen voor. Daarentegen maak ik gebruik van een timer interrupt service routine. In deze ISR had ik decrementeerende software counters. Elke milliseconde, centiseconde, deciseconde en seconde (ja, dit zijn legitieme eenheden) worden er een of meerdere counters gedecrementeerd, mits hij groter is dan 0. In voorbeeld laat ik ledPin 13 knipperen.

```
if(!timer) {           // als timer niet 0 is...
    timer = interval;  // zet een waarde in timer
                      // timer decrementeert in ISR
    TGL(PORTB, ledPin); // ledPin is hier 5
}
```

De if(!timer) en timer = interval gebruikte ik heel erg veel op exact deze wijze. Toen begon ik ze in een lijn te schrijven. Dit wordt niet geaccepteerd door de meeste programmeurs. Hiermee kan je 'goede' programmeurs ook altijd erg boos mee maken. Doen dus!

```
if(!timer) { timer = interval;
    TGL(PORTB, ledPin); // ledPin is hier 5
}
```

Ik kan hier dus mee leven, sommigen kunnen dat niet. Ik ging toen dus denken, dit moet toch tot een macro te verwerken zijn? En ja dat kon en nee het werd niet duidelijker. Dat zag er zo uit:

```
expired(timer, interval) {
    TGL(PORTB, ledPin); // ledPin is hier 5
}
```

Wij wisten al wat dit precies deed, maar stel nou dat je nog niet wist wat we hier nou precies deden? Dan zou je het niet snappen zonder eerst die macro op te zoeken. Het is geen standaard C code en eigenlijk helemaal onleesbaar.

```
if(!timer) {           // als timer niet 0 is...
    timer = interval; // zet een waarde in timer
```

Dit is wel standaard C code en iedereen begrijpt wat dit doet. Dit is daarom beter om het zo te laten staan.

De eerste theorie gaat dus wel op in dit voorbeeld. Maar nu...

Als je complexe processen ga coderen, wil je gebruik maken van een state-machine zoals in het vorige hoofdstuk. Ik heb van een collega de goede kanten van macro's leren kennen en met die kennis heb ik voor onze state-machines een zevental macro's ontwikkeld die onze state-machine nog duidelijker maken.

Een state zoals ik ze gebruik op werk:

```
stateFunction(nameOfState) {
    entryState {
        // wordt eenmalig uitgevoerd
    }

    onState {
        // wordt continu uitgevoerd
        if(/* statement */) exitFlag = true;
    }

    exitState {
        // wordt eenmalig uitgevoerd bij het verlaten van de state
        return true; // signaal naar state-machine toe
    }
}
```

'State', 'entryState', 'onState' en 'exitState' zijn alle vier macro's. Zonder te laten zien hoe deze macro's er uit zien, ga ik de state-functie invullen met instructies.

Als ik nu de open crossing state-functie omzet naar een format met deze macro's dan komt hij er als volgt uit te zien:

```
stateFunction(openCrossing) {  
  entryState {  
    digitalWrite(led1, HIGH);  
    Serial.println("led 1 wordt nu eenmalig aangezet");  
  }  
  onState {  
    blinkLeds(50);  
    controlArms(-1);  
    if(angle == 0 || occupied.track1 || occupied.track2) {  
      exitFlag = true;  
    }  
  }  
  exitState {  
    digitalWrite(led1, LOW); // zet alle lampen uit  
    digitalWrite(blinkLed1, LOW);  
    digitalWrite(blinkLed2, LOW);  
    return true; // signaal naar state-machine toe  
  }  
}
```

Door de macro's wordt de code iets beter uitgelijnd en beter afgebakend. Je kan in een oogopslag ook de drie sub-states waarnemen. De code neemt ook iets minder ruimte in beslag

Het heeft even geduurd maar ik ben nu pas tevreden met onze state-functies. Er staat bijna niets in wat we nog niet gezien hebben. Alleen de exitFlag is nieuw, maar ik denk dat je nu wel kan raden waar deze goed voor is (het uitvoeren van de exitState).

Ik wil nu een trein laten pendelen tussen twee secties. We beginnen met een state te maken die de trein naar het station stuurt.

Stel nu dat we een trein laten rijden in een richting door `digitalWrite()` te gebruiken totdat de trein over een lichtsluis rijdt. De trein mag er tevens niet te lang over doen, dan is hij uitgevallen of heeft hij onderhoud nodig.

Als ik al deze voorwaardes in de state wil zetten, dan ziet de state er zo uit:

```
stateFunction(treinNaarStation) {
  entryState {
    Serial.println("De trein gaat nu rijden");
    digitalWrite(trackPower, HIGH);
    digitalWrite(directionPin, HIGH);

    timeout = 50; // de trein heeft 50 seconde
  }
  onState {
    if(digitalRead(lichtSluis)) {
      exitFlag = true;
    }
    if(!timeout) { // zelfde als timeout == 0
      treinStatus |= TE_LAAT;
      exitFlag = true;
    }
  }
  exitState {
    digitalWrite(trackPower, LOW);
    Serial.println("Baan is afgeschakeld");

    if(treinStatus & TE_LAAT) {
      Serial.println("de trein is te laat!");
    }
    else {
      Serial.println("de trein is op tijd aangekomen");
    }
    return true; // einde state
  }
}
```

Ik hou code leesbaar door het in hap klare brokjes te serveren. Laten we nu eens deze state-functie analyseren en kijken wat alles doet.

In de `entryState` print ik eenmalig een bericht op de monitor, ik zet de `directionPin` en de `trackPower` pin hoog. En ik stel een `timeout` in op 50 seconde.

In de onState doen we twee dingen. We kijken continu of de trein over de lichtsluis rijdt en we kijken of timeOut 0 bereikt. Als een van deze condities waar is, dan zetten we de exitFlag. Dit leidt er toe dat exitState eenmalig wordt uitgevoerd en deze hele State wordt daarna niet meer uitgevoerd. Als de trein te laat is dan zet ik een vlag (TE\_LAAT) in de status variabele (treinStatus) door middel van de |= instructie.

In de exitState printen we dat de baanspanning is afgeschakeld en als de TE\_LAAT vlag waar is, dan printen we ook dat de trein te laat is en anders printen we dat de trein op tijd is aangekomen.

Dit vind ik dus een goed voorbeeld dat macro's ook de boel kunnen verduidelijken. De macro's zijn bovendien getest of ze werken en deze zijn ook niet in staat om vreemd gedrag te veroorzaken. Of ze nu wel of niet type-safe zijn, is niet relevant. Ze werken zoals ze moeten werken.

## §2.4 De macro operators

### §2.4.1 De enkele # macro operator

Voordat ik naar het volgende hoofdstuk spring, wil ik eerst nog de laatste macro truuks laten zien. In macro's mogen we de enkele # en de dubbele ## macro operators gebruiken. Ze doen elk iets anders.

```
#define printName(x) Serial.println(#x);
```

Deze enkele # zet in dit voorbeeld "aanhalingstekens" om dat wat we er in voeren. In hoofdstuk 1 hadden we deze enum voor onze state namen:

```
enum states {  
    crossingIdle,  
    closeCrossing,  
    openCrossing,  
};
```

Met de macro kunnen we deze namen printen op het scherm zoals ze zijn. Deze regel:

```
printName(closeCrossing);
```

print het woord "closeCrossing" op het scherm voor ons. Als we hem helemaal uitschrijven, krijgen we:

```
Serial.println("closeCrossing");
```

We kunnen deze macro operator gebruiken om elke keer een state naam op het scherm te printen wanneer er een nieuwe state wordt uitgevoerd zonder veel extra tik werk.

### §2.4.2 De dubbele ## macro operator

De dubbele ## operator doet iets compleet anders voor ons. Deze macro operator plakt hetgeen wat achter de ## staat vast aan hetgeen wat voor de ## operator staat.

```
#define callFunction(x) x##State()
```

Als we deze macro combineren met een van onze constanten mogen we tikken:

```
callFunction(openCrossing);
```

Als we deze macro uitschrijven, krijgen we:

```
openCrossingState();
```

Dit is slechts een functie call naar de functie genaamd 'openCrossingState'. Je ziet hier dat 'State()' achter openCrossing (de x) wordt geplakt.

Deze macro operators gaan we nodig hebben om de code van onze state-machine visueel op te knappen zonder dat we de werking aanpassen.

In de volgende paragraaf laat ik zien de gebruikte macro's er uit zien.

## §2.5 De state-machine macro's

### §2.5.1 De stateFunction macro

Dit is de macro voor de state-functies.

```
#define stateFunction(x) static bool x##F(void)
```

Deze is niet bijzonder ingewikkeld. 'stateFunction(stateNaam)' is slechts een vervanging voor:

```
static bool stateNaamF(void)
```

### §2.5.2 De entryState macro

```
#define entryState if(runOnce)
```

De entryState { uitgevouwen wordt:

```
if(runOnce) {
```

runOnce is een vlag die we al ontmoet hebben. En deze vlag wordt ge'clear'ed nadat de entryState code is uitgevoerd. Dit zorgt er voor dat de entryState slechts 1 malig wordt uitgevoerd.

### §2.5.3 De onState macro:

```
#define onState runOnce = false;
```

De onState { uitgevouwen, is een beetje apart. Dit wordt dan:

```
runOnce = false; {
```

Dan kunnen we accolade er onder zetten voor wat duidelijkheid

```
runOnce = false;  
{
```

De onState macro is hier degene die de runOnce vlag clear't. In C/C++ mag je namelijk ten alle tijden extra accolades gebruiken tussen code regels. Dit kan je doen om een bepaalde variabele scope te realiseren. Of om een niet-standaard macro constructie te laten werken. Let op dat het clearen van de runOnce flag nu elke programma cyclus gebeurt. Dit verschijnsel noemen we overhead. Feitelijk laat ik de Arduino meer (onnodig) werk verrichten om slechts een klein beetje leesbaarheid te realiseren.



#### §2.5.4 De exitState macro:

```
#define exitState if(!exitFlag) return false; else
```

De exitState macro laat zien wat we nu precies doen met die exitFlag. Ik zal hem ook meteen netjes formatteren.

```
if(!exitFlag) return false;
else {
    // exit state code
    return true;
}
```

Een functie die een variabele retourneert, moet dit wel verplicht altijd doen. Tot nu toe hebben we gezien dat de exitState 'true' retourneert, maar wat als de state nog niet klaar is en de exitState helemaal niet wordt uitgevoerd? De state-functie moet wel iets retourneren. En hij doet dit ook wel. Ik heb het alleen verborgen met deze macro.

Zolang de exitFlag 'false' is, wordt 'false' geretourneerd naar de state-machine. De state-machine snapt hierdoor dat de state nog niet klaar is en zal deze state blijven aanroepen. Deze blijft dit doen net zolang totdat exitFlag 'true' is en de else body zal uitvoeren. In ons geval is de else body onze exitState code.

### §2.5.5 STATE\_MACHINE\_BEGIN en STATE\_MACHINE\_END

De state-machine zelf heeft in zijn uiteindelijke vorm nog wat code wat naar mijn mening ook wegge'macro't kon worden. Elke state-machine begint altijd met dit stukje:

```
extern void pendelStates(void) {
    if(!enabled) {
        if(!pendelStatesT) { // heeft een unieke naam per state-machine
            enabled = true;
        }
    }
    else switch(state){
    default: Serial.println("unknown state executed, state is idle now");
    case pendelStatesIDLE: return;

        State(x) { //...
```

En eindigt met:

```
        // ...
        nextState(y, 0);
    }
    break;
}
return false;
}
```

Deze code staat er altijd. Deze code geeft details over hoe de state-machine functioneert. Maar hoeven we dit bij elke state-machine die we maken ook te zien? Het antwoord is nee. Ik had daarom nog twee aanvullende macro's ontworpen die onze state-machine er zo uit kan laten zien:

```
// STATE-MACHINE
extern bool pendelStates(void) {
    STATE_MACHINE_BEGIN

    State(x) {
        nextState(y, 0); }

    State(y) {
        nextState(z, 0); }

    State(z) {
        nextState(x, 0); }

    STATE_MACHINE_END
}
```

Het enige wat nu is overgebleven, is dat wat we moeten programmeren. In de state-machine staan nu alleen nog maar de State's en de nextState's. Bovendien kunnen we nu een tab achterwege laten omdat ik een setje {} heb 'verborgen'.

Deze macro's zien er als volgt uit:

```
#define STATE_MACHINE_BEGIN if(!enabled) { \
    if(!pendelStatesT) enabled = true; } \
else switch(state){\
    default: Serial.println("unknown state executed, state is idle now");
state = exampleIDLE; case pendelStatesIDLE: return true;

#define STATE_MACHINE_END break;}return false;
```

De bovenste macro beslaat meerdere lijnen en de onderste slechts 1. Met de for-slash \ kunnen en mogen wij op deze wijze macro's ontwerpen.

### §2.5.6 De State macro.

```
#define State(x) break; case x: if(runOnce) Serial.println(#x); if(x##F())
```

De 'State' blijkt dus een functie te zijn die een boolean type variabele retourneert. Dit is precies wat we eerder gezien hebben in het vorige hoofdstuk.

Als we deze hele macro uitschrijven en netjes formatteren dan krijgen we dit:

```
break;
case stateNaam:
    if(runOnce) Serial.println("stateNaam");
    if(stateNaamF())
```

Dus in plaats van:

```
break;
case stateNaam:
    if(runOnce) Serial.println("stateNaam");
    if(stateNaamF()) {
        nextState(andereState);
    }
```

Hoeven we slechts te tikken:

```
State(stateNaam) {
    nextState(andereState);
}
```

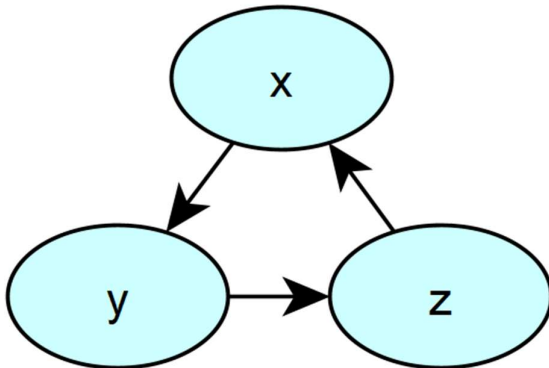
We krijgen nu meer gedaan met minder tikken. Doordat we minder tikken, kunnen we ook minder fouten veroorzaken.

## Hoofdstuk 3 Een programma opbouwen.

### §3.1 state-diagram

Als we nu vanuit scratch een programma willen opbouwen, beginnen we natuurlijk eerst met een state-diagram. Dit kan op de computer, maar je kan er ook 1 krabbelen op een stuk papier.

Voor nu gebruik ik dit voorbeeld:



Ik noem hem 'generator' bij gebrek aan iets beters. Ik ga nu alle componenten van het hele programma toevoegen in volgorde.

Je kan deze code direct kopiëren en plakken in een leeg .ino bestand. Laten we nu kijken of we het echt in 1 keer goed kunnen doen.

### §3.2 #include

Het allereerste wat we gaan doen is het includen van timers.h met de "aanhalingstekens". Deze aanhalingstekens geven aan dat deze bestanden in de project folder liggen. timers.h en timers.cpp moeten we ook nog maken. Dit doen we als het hoofdbestand af is. Het is voor nu belangrijk dat je de volgende werkwijze snapt en zelf kan reproduceren.

```
#include "timers.h"
```

### §3.3 de Macro's

Als tweede zet ik alle macro's in het programma en voor STATE\_MACHINE\_BEGIN gebruik ik ook de naam 'generatorT' voor de timer. Deze code kan je zo kopiëren en plakken naar je project.

```
// MACROS
#define stateFunction(x) static bool x##F(void)
#define entryState if(runOnce)
#define onState runOnce = false; if(!runOnce)
#define exitState if(!exitFlag) return false; else
#define State(x) break; case x: if(runOnce) Serial.println(#x); if(x##F())
#define STATE_MACHINE_BEGIN if(!enabled){\
    if(!generatorT)enabled = true;}\
    else switch(state){\
    default:\
        Serial.println("unknown state executed, state is idle now");\
        state = IDLE;\
    case IDLE: return true;
#define STATE_MACHINE_END break;}return false;
```

De software timer die we gaan gebruiken heet nu generatorT. Hij bestaat nog niet maar dat komt nog. Kijk vooral niet teveel naar de macro's, vergeet niet dat we ze gaan gebruiken en niet willen lezen.

### §3.4 de States (constanten)

We moeten de states aanmaken. Dit doe ik met een enum, maar je mag ook const int of #define gebruiken. Echter omdat de daadwerkelijke waarden van de constanten geenzins uitmaken, is een enum de meest logische keuze. Ik voeg ook een extra IDLE state toe. Aanvankelijk maak ik er geen gebruik van maar we kunnen er later nog wel iets mee doen.

```
// STATES
enum generatorStates {
    IDLE,
    x,
    y,
    z
};
```

### §3.5 de variabelen

Dan moeten we de variabelen aanmaken die we nodig gaan hebben.

```
// VARIABLES
static unsigned char state = x; // we beginnen met state x
static bool enabled = true, runOnce = true, exitFlag = false;
```

Let op dat alle variabelen ook goed geïnitieerd zijn. Als we straks onze state-machine gaan aanroepen dan wordt nu als eerste state x uitgevoerd.

### §3.6 de Functies

Dan moeten we de functie nextState() in het programma plaatsen, die 'generatorT' als timer gebruikt.

```
// FUNCTIONS
static void nextState(unsigned char _state, unsigned char _interval) {
    runOnce = true;
    exitFlag = false;
    state = _state;

    if(_interval) {
        enabled = false;
        generatorT = _interval;
    }
}
```

### §3.6 De state-functies

Nu we alle ondersteunende elementen hebben toegevoegd, gaan we de state-functies en de state-machine toevoegen. We gaan nog niks invullen voor de state-functies, we gaan slechts lege state-functies toevoegen. Wat we wel invullen, zijn de blokken van de entryState, onState en exitState te samen met de exitFlag.

Die exitFlag voegen we ook alvast toe, zodat we hem nooit kunnen vergeten. Als zou je de exitFlag vergeten de zetten, dan zou de state-machine in een bepaalde state kunnen blijven hangen.

```
// STATE FUNCTIONS
stateFunction(x) {
    entryState {

    }
    onState {

        exitFlag = true;
    }
    exitState {
        return true;
    }
}

stateFunction(y) {
    entryState {

    }
    onState {

        exitFlag = true;
    }
    exitState {
        return true;
    }
}
```

```

stateFunction(z) {
  entryState {

  }
  onState {

    exitFlag = true;
  }
  exitState {
    return true;
  }
}

```

### §3.7 de state-machine

En natuurlijk de state-machine:

```

// STATE MACHINE
extern bool generator(void) {
  STATE_MACHINE_BEGIN

  State(x) {
    nextState(y, 0); }

  State(y) {
    nextState(z, 0); }

  State(z) {
    nextState(x, 0); }

  STATE_MACHINE_END
}

```

Omdat elke state slechts naar 1 andere state kan springen heeft deze state-machine zelf geen invulwerk nodig. In principe is deze *good to go*. Optioneel kan je een paar van die nulletjes vervangen door een ander getal voor een 'inter-state delay'. Let er wel op je geen waardes invult die hoger zijn dan 255.

### §3.8 void setup()

Het enige wat nu nog ontbreekt, zijn de void setup(), void loop() en de bestanden timers.cpp en timers.h

In de setup doen we slechts de timers en de seriele communicatie initialiseren. De timers initialiseren we met een functie genaamd initTimers(). Deze maken we nog aan in timers.cpp

```

// SETUP
void setup() {
  initTimers();
  Serial.begin(115200);
  Serial.println("arduino working");
}

```



### §3.9 void loop()

De void loop heeft nu slechts een taak en dat is het aanroepen van onze state machine 'generator'.

```
// LOOP
void loop() {
    generator();
}
```

### §3.10 timers.cpp & timer.h

Om het project te kunnen compileren, is het nodig dat we nu de timer bestanden maken. Maak via de windows verkenner (of je tekst editor) twee tekstbestanden aan en noem ze timers.cpp en timer.h. Kopieer de volgende code in timers.cpp

```
#include <Arduino.h>
#include "timers.h"

extern void initTimers() {
    cli();
    TCCR2B = 0;
    TCNT2 = 0;
    OCR2A = 249;
    TCCR2A |= (1 << WGM21);
    TCCR2B |= (1 << CS20);
    TCCR2B |= (1 << CS21);
    TCCR2B &= ~(1 << CS22);
    TIMSK2 |= (1 << OCIE2A);
    sei();
}

volatile unsigned char generatorT;

ISR(TIMER2_COMPA_vect) {
    static unsigned char _1ms, _10ms, _100ms;
    // 1ms timers
    _1ms += 1;

    // 10ms timers
    if(!(_1ms % 10)) { _1ms = 0; _10ms += 1;
        if(generatorT) generatorT--;
    }

    // 100ms timers
    if(!(_10ms % 10)) { _10ms = 0; _100ms += 1;

    //1000ms timers
    if(!(_100ms % 10)) { _100ms = 0;

    }
    }
    }
}
```

En kopieer deze regels in timers.h

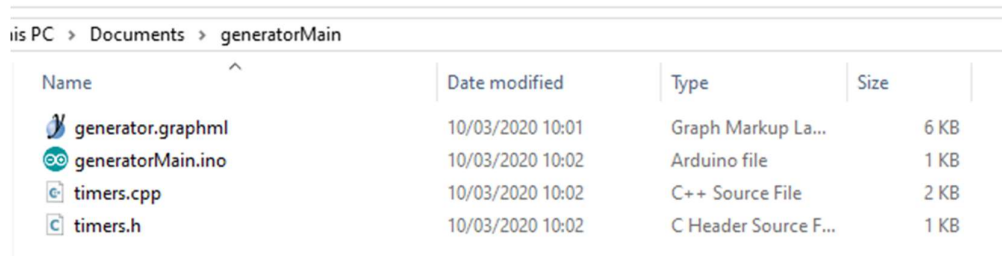
```
extern void initTimers();  
  
extern volatile unsigned char generatorT;
```

Uitleg van deze code staat in de bijlage.

### §3.11 Eerste tussentijdse test

Nadat je deze bestanden aangemaakt, ingevuld en opgeslagen heb, is je programma compileerbaar. Probeer het uit en klik op 'verify sketch' in de Arduino IDE om te compileren.

Vergeet ook niet om je .ino bestand een naam te geven en op te slaan. Ik heb de mijne 'generatorMain.ino' genoemd. Mijn project folder ziet er nu zo uit:

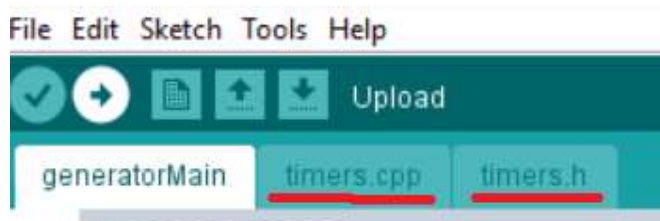


C:\Users\... \Documents > generatorMain				
Name	Date modified	Type	Size	
generator.graphml	10/03/2020 10:01	Graph Markup La...	6 KB	
generatorMain.ino	10/03/2020 10:02	Arduino file	1 KB	
timers.cpp	10/03/2020 10:02	C++ Source File	2 KB	
timers.h	10/03/2020 10:02	C Header Source F...	1 KB	

generator.graphml is het bollendiagram die ik heb gemaakt op de computer.

Arduino heeft een aparte eigenschap. Het .ino bestand moet dezelfde naam hebben, als de folder waarin deze zit.

Als je de Arduino IDE nu opnieuw opstart, dan zie je de timer bestanden ook verschijnen als tabbladen.



Je kan je programma ook al uploaden, alleen met de nog lege state-functies doet het nog niet zo veel. Wat wel doet is herhaaldelijk de states x, y en z onder elkaar printen. Dit zit in de 'State' macro gebakken. Zo kunnen we in ieder geval waarnemen dat de state-machine werkt.

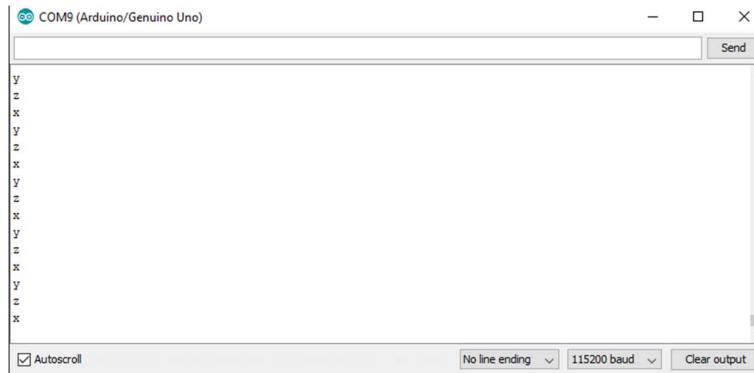
Helaas kunnen we nooit alle bugs voorkomen. Voor die bugs waarvoor ik je niet kan behoeden, kan ik je wel helpen ze te lokaliseren.

Op dit punt is het raadzaam om te kijken of de state-machine inderdaad ook echt werkt. Ik zeg wel dat hij werkt, maar heb ik eigenlijk wel gelijk? Mijn vrouw zegt altijd van niet.

Stel nu dat we eerst de state-functie zouden invullen met code en er daarna achter komen dat er ergens in het programma een bug zit. Dan weten we niet precies waar we moeten zoeken. Als wij nu bevestigen dat deze state-machine werkt, dan weten we in ieder geval zeker dat deze werkt. Als we daar zeker van zijn, kunnen we de state-functies gaan invullen met code en die 1 voor 1 testen.

Het tussentijds testen is een zeer handige methode om bugs te vinden voordat ze een probleem kunnen vormen.

Ook doordat we de states op het scherm printen, weten we in welke state een eventuele bug zit. Dan weten we waar we moeten zoeken. En omdat de state-functie is opgedeeld in de drie kleinere sub-states zou het je relatief weinig moeite moeten kosten, om de bug te vinden.



Als je hebt bevestigd dat de state-machine keurig alle states doorloopt, kunnen we iets gaan doen met de nog lege state-functies.

### §3.12 state-functies invullen

Met een kaal Arduino board kan men slechts twee dingen doen. We kunnen ledPin 13 iets laten doen en we kunnen tekst sturen en ontvangen met de seriële monitor. Laten we nu beiden doen.

Om de fijne werking van de state-machine structuur te demonstreren, lijkt het mij leuk om elke state-functie iets te laten printen in de entryState en de exitState en om ledPin 13 te laten knipperen gedurende een bepaalde tijd. Voor state x laat ik de led 10x toggelen met een interval van 1s.

```
stateFunction(x) {
    static uint8_t blinkCounter;

    entryState {
        Serial.print("entering state x ");
        Serial.println("blinking led 13 10x");
        blinkCounter = 0;
    }
    onState {
        if(!generatorT) {          // als timer 0 is
            generatorT = 100;      // zet timer op 1 seconde
            digitalWrite(ledPin, ~digitalRead(ledPin)); // knipper
            blinkCounter++;         // verhoog teller
            Serial.println("blink");
        }

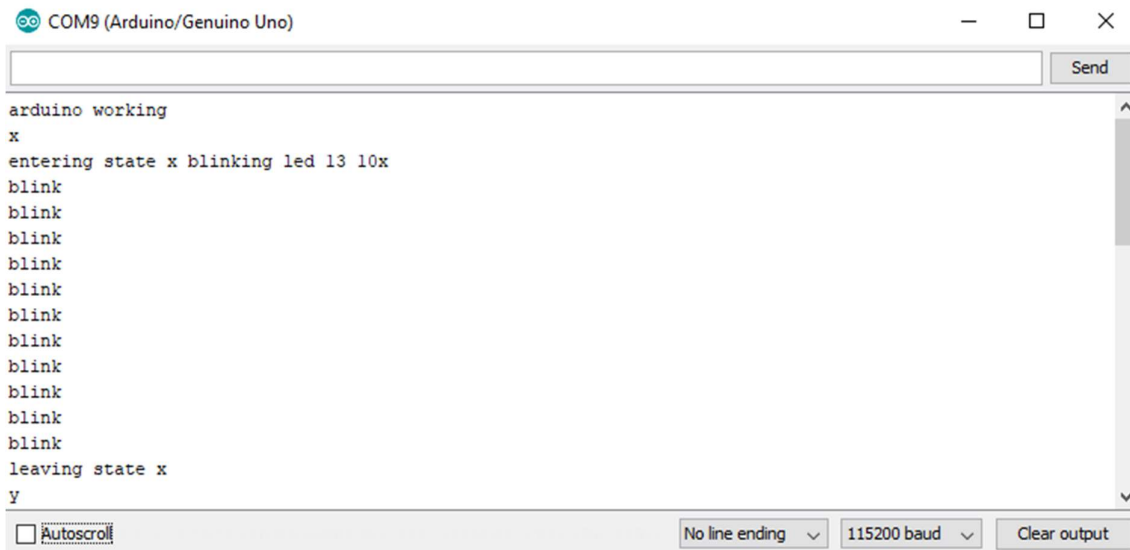
        if(blinkCounter > 10) exitFlag = true; // 10x geknipperd is exit
    }
    exitState {
        Serial.println("leaving state x ");
        digitalWrite(ledPin, LOW);
        return true;
    }
}
```

Ik gebruik een static variabele (static variabele houdt zijn waarde vast) om bij te houden hoe vaak er geknipperd wordt.

Voordat dit werkt moeten we ledPin #definieren als 13 en de pinMode instellen. De pinMode instructie zou in dit specifieke geval ook nog in de entryState kunnen zetten. Dit hoeft niet perse in de void setup(). Het enige nadeel is dat de pinMode instructie wordt uitgevoerd elke keer dat state x wordt uitgevoerd. Dit kan echter geen kwaad. De #define voor deze led pin hoeft ook niet perse boven aan het bestand te staan. Je kan deze ook boven de stateFunction zetten bijvoorbeeld.

```
#define ledPin 13
stateFunction(y) { ...
    entryState {
        pinMode(ledPin, OUTPUT);
```

We kunnen deze state functie alvast testen, laten we kijken of de teksten op het scherm verschijnen zoals we verwachten.



Dit ziet er goed uit. De led knippert ook goed mee.

Voor state y wil ik iets doen met user input. Ik wil door middel van de terminal zelf de led kunnen laten knipperen en de state beeindigen.

```
stateFunction(y) {
  entryState {
    Serial.println("entering state y");
    Serial.println("H = led on");
    Serial.println("L = led off");
    Serial.println("T = led toggle");
    Serial.println("Q = exit");
  }
  onState {
    if(Serial.available()) {
      byte serialByte = Serial.read();
      switch(serialByte) {
        case 'H': digitalWrite(ledPin, HIGH); break;
        case 'L': digitalWrite(ledPin, LOW); break;
        case 'T': digitalWrite(ledPin, !digitalRead(ledPin)); break;
        case 'Q': exitFlag = true; break;
      }
    }
  }
  exitState {
    Serial.println("leaving state y ");
    digitalWrite(ledPin, LOW);
    return true;
  }
}
```

Ik kijk met Serial.available() of er iets in te lezen valt, dan lees ik de byte in en voer de waarde daarvan in de switch-case.

Als alles goed gaat, zou je zoiets moeten zien:



```
COM9 (Arduino/Genuino Uno)
Send
blink
blink
leaving state x
y
entering state y
H = led on
L = led off
T = led toggle
Q = exit
leaving state y
z
entering state z
leaving state z
x
entering state x blinking led 13 10x
blink
Autoscroll No line ending 115200 baud Clear output
```

Ik kon met de letters; 'H', 'L' en 'T' de led aansturen en met 'Q' kon ik de exitFlag zetten. Ik had ook al een paar tekstjes toevoegd aan state z ("entering state z" en "leaving state z").

Op dit punt raad ik aan om voor jezelf wijzigingen aan deze code te brengen. ledPin 13 is een PWM pin dus je kan hem ook laten faden. Probeer zelf ook eens om een nieuwe state toe te voegen om de flow aan te passen. Je kan ook een nieuwe timer toevoegen aan timers.cpp en timers.h. Probeer eens om de tijdbasis van generatorT op 100ms te zetten.

### §3.13 Herhaalbaarheid.

Als je een nieuw project wilt opzetten, wil je dit snel kunnen doen. Wat je zeker niet wilt doen, is telkens dezelfde dingen opnieuw en opnieuw doen. We hebben nu het wiel eindelijk eens goed rond gekregen.

Je kan voor jezelf een project aanmaken met een kale state-machine met misschien 1 of 2 lege state-functies zoals is beschreven in dit hoofdstuk. Als je dan een nieuw project wilt beginnen of een bestaand project wilt omcoderen naar deze nieuwe structuur dan hoeft je slechts een kopie te maken van het 'lege' project.

Je hebt dan het voordeel dat de grondslag al gelegd is. Het enige wat je dan nog hoeft te doen, is je state-machine uit te breiden en in te vullen. Je hoeft niet meer na te denken over wat je allemaal nog zou moeten toevoegen.

Je werkt dan geforceerd met een structuur die is uitontwikkeld en geperfectioneerd. Deze structuur staat je toe, om complexe processen om te coderen naar werkende code. Die misschien wel bug vrij is in de eerste poging.

## Hoofdstuk 4 multi-taken

Een arduino kan zonder moeite meer dan 1 state-machine aanroepen. Je kan een Arduino zonder problemen 2 ahobs laten afhandelen. Dan moet de arduino 6 verschillende state-machines aanroepen.

Tot nu toe hebben we code behandeld waarin slechts 1 state-machine zit in het .ino bestand. Als we meerdere state-machines willen toevoegen aan dat bestand, moeten we alle gebruikte variabelen dupliceren. Er moet een 2<sup>e</sup> 'state', 'runOnce' en 'exitFlag' worden aangemaakt en ze mogen niet zomaar dezelfde naam hebben.

Dit probleem kan je op menig manier aanpassen. Je kan gebruik maken van een C++ functionaliteit genaamd 'namespace', je kan structs aanmaken voor deze variabelen. Of we geven elke state-machine een eigen .cpp en .h bestanden.

Dit laatste heeft mijn voorkeur om meer dan 1 rede. De bestanden blijven kleiner en we hoeven geen syntax aan te passen.

### §4.1 code opsplitsen in meerdere bestanden.

Als je nog nooit met meerdere bestanden heb gewerkt dan heb je waarschijnlijk ook al ontdekt dat je .ino tot een gigantische draak kan uitgroeien waarbij je ontzettend veel moet scrollen. Om onder andere wat duidelijkheid voor ons zelf te maken, kunnen we veel van onze functies en variabelen verhuizen naar andere bestanden.

Ook voor de state-machine structuur zoals ik hem bedacht heb, willen we ook met meerdere bestanden werken.

#### §4.1.1 static & extern

static en extern zijn twee belangrijke keywords in C. Ze zijn er om de scope van zowel functies als variabelen aan te geven. Het is nodig om deze woorden te leren gebruiken voordat we met meerdere bestanden gaan spelen. Ze zijn enigszins ingewikkeld omdat je ze niet altijd hoeft te tikken en omdat ze voor functies anders werken dan voor variabelen. Als klap op de vuurpijl kan het ook nog per compiler verschillen. Gelukkig is de door Arduino gebruikte compiler vrij simpel hier in. Als je de uitleg niet helemaal snapt, is dat niet erg. In de volgende paragraaf geef ik er voorbeelden bij.

In principe is het voor functies nooit nodig om noch static noch extern te tikken. Ik moet hierbij vermelden dat als je niks tikt voor een functie, dat deze standaard extern is. Dat wil zeggen deze functie aangeroepen mag worden vanuit andere bestanden.

Het is mijn gewoonte om alsnog extern en static te gebruiken ook wanneer het niet nodig is. Als een functie slechts wordt aangeroepen in 1 bestand dan tik ik er static voor. Dat is misschien niet nodig, maar zo kunnen ik, jij en andere collega's in een oogopslag zien, wat de scope is van de functie.

Voor variabelen is het vaak wel nodig wel om deze keywords te gebruiken. Als je niks voor een variabele tikt dan is hij standaard static. Als je met meerdere bestanden werkt en een variabele wilt hebben die toegankelijk is tot meerdere bestanden dan is het nodig om in een .h bestand deze variabele te declareren met extern. Alle bestanden die deze .h includen, kunnen gebruik maken van deze variabele. Echter er moet ook een source (.cpp) bestand zijn die ook deze variabele declareert maar dan zonder extern.

Met extern vertellen wij eigenlijk aan de compiler: "deze variabele bestaat echt, ik beloof het".

```
extern unsigned int x; // in een header bestand
```

En met de declaratie zonder extern vertellen wij de compiler: "Ik wil dat je hier en nu geheugen toekent aan deze variabele".

```
unsigned int x; // in een source bestand
```

Je kan een externe variabele declaratie beschouwen als een functie prototype maar dan voor variabelen. Om deze rede kan je bij een externe declaratie geen waarde initialiseren. Je mag niet tikken:

```
extern int x = 4;
```

In deze cursus hebben we ook al gezien dat we static ook nog op een andere manier kunnen gebruiken. Namelijk in een functie voor lokale variabele. In dit specifieke geval zorgt static ervoor dat de lokale variabele zijn waarde blijft behouden tussen functie calls door. Je mag hem in zo'n geval ook nog een initiele waarde meegeven.

```
void foobar(){
    static unsigned int x = 4; // in een source bestand
    x++;
    Serial.println(x); // print 5, 6, 7 etc
}
```

Als je in een functie voor een lokale variabele extern tikt, dan geef je aan dat die variabele een globale variabele is die elders is gedeclareerd. Dat wil zeggen dat je eigenlijk een nutteloze instructie uitvoert, want je mag in die functie namelijk ook gewoon dezelfde globale variabele gebruiken zonder hem extern te declareren. Dit is alleen handig als je in een functie gebruik wilt maken van een lokale en een globale variabele die dezelfde naam hebben. Maar aan deze praktijken gaan we niet beginnen. Zoals ik het beschrijf, is dit als "*shooting yourself in the foot*".



#### §4.1.2 source en header files

Nu laat ik met code voorbeelden zien hoe je static en extern kan en moet gebruiken. Stel nu dat we voor ons .ino bestand 2 aanvullende bestanden maken. We noemen ze foobar.cpp en foobar.h. foobar.cpp krijgt 2 functies genaamd 'hello' en 'world' waarvan alleen 'hello' extern is. foobar krijgt ook 3 variabelen x, y en z waarvan x en y globaal zullen zijn en z static of lokaal. foobar.cpp:

```
int x, y;    // x en y zijn globaal, we mogen geen extern tikken
static int z = 5; // static is ook hier niet nodig.

extern void hello() { // extern is niet nodig
    Serial.print("hello ");
    world(); // functie call naar world
}

static void world() // static is ook niet nodig
    Serial.print("hello");
}
```

Waar je nu op moet letten is dit: functie hello doet een call naar world. World ligt echter onder hello en daarvoor moeten we een functie prototype maken of we moeten de posities omwisselen. Omdat we nu met source en header bestanden bezig zijn, kies ik er voor om beide functie prototypes in de header te plaatsen.

foobar.h komt er nu zo uit te zien:

```
// z en y hoeven en mogen hier niet staan
extern int z; // overal toegankelijk, extern is nodig

extern void hello(); // overal toegankelijk, extern is overbodig
static void world(); // beperkt tot foobar, static is nodig
```

Ik had de onderste regel, de functie prototype van world, ook in het source bestand mogen tikken. Dan was static niet nodig geweest. Ik had ook hello en world van plaats kunnen wisselen.

Dit is een keuze waar je vrij ben. Meesten vinden dat static functies eigenlijk niet thuis horen in een header file. De static variabelen staan er immers ook niet in. Ik probeer zelf altijd de posities van de functie zo te kiezen dat er geen functie prototypes nodig zijn voor static functies. Ik had in dit geval de plaats van hello en world moeten omwisselen.

Als je nu een groot programma heb, kan je de minder belangrijke delen in functies plaatsen en die functies verhuizen naar een .cpp bestand en een .h bestand. Zo wordt je .ino bestand een stuk kleiner. In mijn .ino bestanden staan er ongeveer maar ~10 regels code in getikt.

#### §4.1.4 source en header files in combinatie met classes

In paragraaf §5.6 Classes van het vorige deel is de class uitgelegd. Bijna elke Arduino library maakt gebruik van classes. Een van de voordelen is dat we bij classes geen static of extern meer nodig hebben. Classes hebben namelijk hun eigen woorden, private en public, voor hetzelfde doeleind. Gelukkig zijn classes voor ons heel makkelijk te plaatsen in de source en header files.

De classe in zijn geheel moet in het header bestand komen te staan en al het andere in het source bestand.

```
class Morse // de classe
{
public:
    Morse(int pin); // constructor
    void dot();     // public 'method'
    void dash();    // public 'method'
    int publicVar;  // public 'variabele'
private:
    int _pin;       // private variabele
    void someFunction(); // private 'method'
};
```

Deze code komt in de header en all het andere code gaat de .cpp in. Om in je .ino gebruik te kunnen maken van de functionaliteiten van Morse moet je morse.h #includen.

#### §4.1.5 #include

Elk source bestand heeft altijd een header bestand nodig als je inhoud van dat bestand beschikbaar wilt hebben voor andere bestanden. Omgekeerd heeft niet elke header bestand een source bestand nodig. Je kan bijvoorbeeld denken aan header bestanden die alleen maar uit #defines bestaan.

De #include regel is net als de #define een onderdeel van de C pre processor. Een #include opent het opgegeven bestand en vouwt dit bestand open op de plaats van de #include. Wat we dan feitelijk doen, is het ophalen van allerlei functie prototypes, externe variabelen en classes die in de headers staan. Ik zal dit visualiseren met een simpel test programma. Stel we hebben:

```
#include "morse.h"

int main () {
    Morse obj1(3);

    obj1.dot();
    obj1.dash();

    return(0);
}
```

Een origineel C++ programma. Als ik handmatig de C pre processor er over heen haal dan krijg ik de volgende code gegenereerd:

```

class Morse
{
    public:
        Morse(int pin);
        void dot();
        void dash();
        int publicVar;
    private:
        int _pin;
        void someFunction();
};

int main () {
    Morse obj1(3);

    obj1.dot();
    obj1.dash();

    return(0);
}

```

Je ziet dat op de plaats van de #include de gehele inhoud van morse.h is geplaatst.

Een #include kan je met <libraryX.h> en met "libraryY.h" uitvoeren. Als je de pijlen gebruikt, gaat de Arduino zoeken naar de opgegeven library in de library folder van Arduino. Als je de aanhalingstekens gebruikt, dan gaat Arduino zoeken in dezelfde folder als waar de .ino staat. Omdat morse.h in deze folder staat, hebben we de "aanhalingstekens" gebruikt.

Hoe het compilatie proces verloopt, wordt uitgelegd in deel 3. Het is niet bijzonder belangrijk om te weten, maar misschien dat je het interessant vindt. Ik denk wel dat helpt om het include systeem beter te begrijpen.

## §4.2 state-machines opsplitsen in source en header files.

We hebben nu het ingewikkelste achter de rug. Nu we het een en ander weten over extern, static en #include is het tijd om dit toe te passen op de code die we al gemaakt hebben.

We gaan alle code van een state-machine verhuizen naar een .cpp en een .h bestand. In hoofdstuk 3 zijn we geordend bezig geweest met een hele state-machine vanuit scratch op te zetten. Ik maak nu een lijstje maak van alle gemaakte stappen in volgorde:

1. Library includes
2. macros
3. states (de enum)
4. variabelen
5. functies
6. state-functies
7. state-machine
8. setup
9. loop

De variabelen en de functie nextState zijn tot nu toe allemaal static en mogen naar de .cpp. De state-functies en state-machine gaan ook naar de .cpp. Tevens willen we de macro's ook in het .cpp bestand hebben.

In het .h bestand komen, de enum met de state namen (constanten), de functie prototype van de state-machine zelf (we moeten deze kunnen aanroepen vanuit een ander bestand). Dus 1 enum en 1 functie prototype. De enum zou in principe ook in het source bestand kunnen staan. Het wordt nog duidelijk waarom we deze in de header willen hebben staan.

Als we zelf source en header files gaan maken, moeten we zelf de Arduino.h library includen. Met deze library kunnen we onder andere gebruik maken van de Arduino functies zoals digitalWrite(); Zonder deze library snapt de compiler ook niet wat nu precies een 'byte' is. Die staat namelijk ge'typedef't in Arduino.h als een unsigned char. De Arduino IDE #include voor ons deze library in het .ino bestand maar de compiler doet dit niet voor andere bestanden.

Doordat de variabelen state, enabled, runOnce en exitFlag en de functie nextState() allemaal static zijn, mogen we deze namen hergebruiken voor andere state-machines. De scope is slechts beperkt tot het source bestand.

Dit houdt dus in dat we voor alle state-machine die we opzetten exact dezelfde variabelen namen kunnen gebruiken als ze hun eigen bestanden krijgen.

De header van generatorT ziet er dan zo uit:

```
#include <Arduino.h>

// STATES
enum generatorStates {
    IDLE,
    x,
    y,
    z,
};

// STATE-MACHINE PROTOTYPE
extern bool generator(void);
```

In het .cpp bestand moeten we weer timers.h includen, maar ook generator.h. Er is verder geen code gewijzigd, daarom laat ik nu de includes zien en de commentaar regels met 1 a 2 regels code voor het overzicht.

```
// INCLUDES
#include "generator.h"
#include "timers.h"

// MACROS
#define stateFunction(x) static bool x##F(void)
    ...

// VARIABLES
static unsigned char state = x;
    ...

// FUNCTIONS
static void nextState(unsigned char _state, unsigned char _interval) {
    runOnce = true;
    ...

// STATE FUNCTIONS
stateFunction(x) {
    entryState {
        ...

// STATE MACHINE
extern bool generator(void) {
    STATE_MACHINE_BEGIN
    ...
```

Het enige wat we nog hoeven te doen, is het nu bijna leeg geworden .ino bestand aanpassen. We hoeven eigenlijk alleen maar de header te includen.

```
// HEADER FILES
#include "timers.h"
#include "generator.h"

// SETUP
void setup() {
    cli();
    initTimers();
    Serial.begin(115200);
    sei();

    Serial.println("arduino working");
}

// LOOP
void loop() {
    generator();
}
```

### §4.3 State-machines parallel uitvoeren

Een van de handigste features van state-machines is dat we meerdere state machines parallel kunnen uitvoeren. Dit kan volkomen onafhankelijk zijn. We hebben voor de ahob nu nog steeds geen code gemaakt om de bezetmelders in te lezen.

#### §4.3.1 Het principe

Je kan simpelweg twee nieuwe bestanden toevoegen met een nieuwe state-machines en die vervolgens aanroepen vanuit de void loop(). We hebben alle benodigde kennis hiervoor behandeld.

Het .ino bestand zou er dan ongeveer zo uit te komen zien:

```
#include "io.h"
#include "timers.h"
#include "railCrossing.h"
#include "readTrack1.h"
#include "readTrack2.h"

void setup() {
    initIO();
    initTimers();
    Serial.begin(115200);
    Serial.println("arduino working");
}

void loop() {
    readTrack1();
    readTrack2();
    railCrossing();
}
```

In de loop worden 3 state-machines aangeroepen. De state-machine railCrossing heeft afhankelijkheden bij de andere state-machines. readTrack 1 en 2 moeten namelijk aan railCrossing vertellen of er nu wel of geen trein is. De state-machine van een van de readTrack bestanden ziet er zo uit:

```
// STATE-FUNCTIONS
State(monitorSw1Sw2) {
    entryState {
        track1occupied = false;
        priority = 0;
    }
    onState {
        if(digitalRead (SW1) == true) {
            priority |= SW1_MASK;
            exitFlag = true;
        }

        if(digitalRead (SW2) == true) {
            priority |= SW2_MASK;
            exitFlag = true;
        }
    }
    exitState {
        track1occupied = true;
        return true;
    }
}
```

```

State(monitorSw1) {
    entryState {
    }
    onState {
        if(digitalRead(SW1) == true) {
            exitFlag = true;
        }
    }
    exitState {
        return true;
    }
}

State(monitorSw2) {
    entryState {
    }
    onState {
        if(digitalRead(SW2) == true) {
            exitFlag = true;
        }
    }
    exitState {
        return true;
    }
}

// STATE-MACHINE
extern void readTrack1(void) {
    STATE_MACHINE_BEGIN

    State(monitorSw1Sw2) {
        if(priority & SW1_MASK) nextState(monitorSw2, 10);
        if(priority & SW2_MASK) nextState(monitorSw1, 10); }

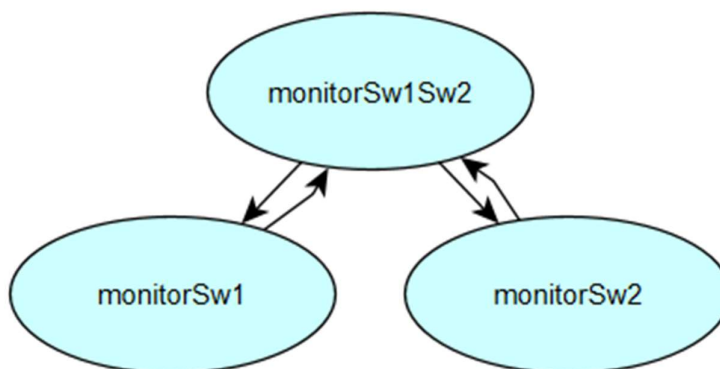
    State(monitorSw1) {
        nextState(monitorSw1Sw2, 250); }

    State(monitorSw2) {
        nextState(monitorSw1Sw2, 250); }

    STATE_MACHINE_END
}

```

De bijbehorende state-diagram:



In de bovensse state houden we continu beide bezetmelders in de gaten. Als detector 1 (Sw1) eerder is waargenomen dan 2, dan wordt de nieuwe state 'monitorSw2' en wordt Sw2 in de gaten gehouden. Wanneer Sw2 wordt gemaakt, wordt de nieuwe state weer 'monitorSw1Sw2'. Gedurende de tijd dat er een trein tussen Sw1 en Sw2 zit, is de flag 'track1occupied' waar. railCrossing weet aan de hand van deze flag of er een trein is of niet. Er zit een vertraging tussen om er voor te zorgen dat de trein genoeg tijd heeft om weg te komen. Ik zal een compleet werkend programma bijvoegen van een ahob op mijn github pagina met een betere structuur voor de bezetmelders.

Elke state-machine heeft hier zijn eigen timer nodig. Je kan er voor kiezen om millis() te gebruiken, of je gebruikt het bijgevoegde timer interrupt en voegt hieraan zelf nieuwe timers toe.

Wat je hier voornamelijk weer ziet, is weer een van die vele state-machines, weer exact dezelfde syntax, weer... hetzelfde en hetzelfde. Ik wil hier duidelijk maken dat het deze "hetzelfde" is die zo enorm belangrijk is. Hoe meer en meer we hetzelfde doen, hoe minder en minder geneigd wij zijn om hetzelfde fout te doen. Het maakt het zo ook makkelijker om onze resultaten te reproduceren.



### §4.3.2 communicatie tussen state-machines

We kunnen de flag 'monitorSw1Sw2' op 2 manieren doorgeven. We kunnen de flag 'track1occupied' globaal maken. Of we houden de flag lokaal en kunnen de status doorgeven met een functie. Hoewel beide methoden kunnen, wordt optie 2 over het algemeen als veiliger beschouwd.

In beide gevallen is het nodig dat ook railCrossing zowel readTrack1.h en readTrack2.h include. Dit is noodzakelijk als we de informatie willen uitwisselen. Beschouw nu deze functie.

```
extern bool track1returnState() {  
    return track1occupied;  
}
```

Deze functie retourneert de waarde van track1occupied. Het is een externe functie en railCrossing kan deze functie aanroepen om te kijken of er een trein is of niet. Eerder hebben we deze code gezien in een oude state-functie van railCrossing:

```
bool openCrossingState(void) {  
    if(runOnce) {  
        runOnce = false;  
        // entryState code  
        digitalWrite(led1, HIGH); // zet de buitenste lampen aan  
        Serial.println("led 1 wordt nu eenmalig aangezet");  
    }  
    // onState code  
    blinkLeds(50);  
    controlArms(-1);  
    if(angle == 0 || occupied.track1 || occupied.track2) {  
        // exitState code  
        digitalWrite(led1, LOW); // zet alle lampen uit  
        digitalWrite(blinkLed1, LOW);  
        digitalWrite(blinkLed2, LOW);  
        return true;  
    }  
    else return false;  
}
```

Hier gebruikten we nog een struct met bitfields (occupied.track1 en occupied.track2) om de status van de sporen aan te duiden.

Nu we de bestanden gaan opsplitsen, is het enigszins lastig deze structuur te handhaven. Het kan wel, maar wat we beter kunnen doen is deze regel:

```
if(angle == 0 || occupied.track1 || occupied.track2) {
```

Te vervangen door:

```
if( angle == 0 || track1returnState() || track2returnState() ) {
```

De state-machines communiceren nu met elkaar door middel van deze externe functies.

Eerder heb ik gezegd dat we de enum met state namen beschikbaar wilde hebben voor andere bestanden. Ik ga nu uitleggen waarom dat is. Om state-machines met elkaar te laten communiceren kunnen we in plaats van vlaggen ook gebruik maken van elkaars 'state' variabelen.

Stel we maken deze functie in readTrack1:

```
extern uint8_t track1GetState() {  
    return state;  
}
```

En we maken dezelfde functie in readTrack2:

```
extern uint8_t track1GetState() {  
    return state;  
}
```

Dan kan railCrossing deze functies gebruiken om te kijken wat de state is van deze 2 state-machines.

Deze functies retourneren de waarde van state. Dit is natuurlijk een getal. Als we zouden tikken:

```
if( track1GetState() == 2) { // wat is 2 nou weer?
```

Dan zou het niet precies duidelijk zijn, maar nu hebben wij de enum van readTrack1 in de header gezet. railCrossing heeft dan ook toegang tot zijn state-constanten. Daarom mogen wij nu tikken:

```
if( track1GetState() == monitorSw1Sw2) { // dit is duidelijker
```

Als we goed kijken wat nu de waarde van 'state' van readTrack1 is wanneer er een trein is. Dan valt het op dat als de state gelijk is aan monitorSw1Sw2 er geen trein is. We zouden er hiervoor kunnen kiezen om de occupied flags te verwijderen en ze te vervangen door de xxxGetState() functies.

In het geval van nested state-machines willen we soms de 'state' variabele kunnen zetten. Of we willen bij een state-machine handmatig ergens midden in beginnen. Als we nu deze functie toevoegen:

```
extern void track1SetState(uint8_t newState) {  
    state = newState;  
    runOnce = true;  
}
```

Dan kunnen we in code tikken:

```
track1SetState(monitorSw1Sw2); // beter dan track1SetState(1);
```

En omdat we nog steeds de constanten van readTrack1 kunnen gebruiken hoeven we ook hier geen nummertje te tikken.

Deze xxxSetState() en xxxGetState() zijn bij mij vaste toevoegingen aan de state-machines. Ik gebruik ze lang niet altijd, maar als ik ze nodig heb dan heb ik ze tot mijn beschikking.

## §5 slotwoord

In dit deel heb je geleerd, hoe je processen kunt omzetten in code op een georganiseerde wijze. Daarbij hebben we de basis implementatie geleerd van een simpele state-machine in de vorm van een switch-case. Deze state-machine hebben we telkens met kleine stapjes verbeterd en verbeterd totdat we op onze uiteindelijke structuur zijn beland.

Ik heb laten zien hoe we deze code nog leesbaarder en gestructureerder kunnen maken met macro's. En hoe we het geheel nog duidelijk kunnen maken door code op te splitsen in meerdere bestanden.

In de bijlage vind je meer details over hoe ik gebruik maak van software timers. Dit is natuurlijk optioneel en je blijft vrij om `millis()` te gebruiken.

Ik heb zelf ondervonden dat de meeste bugs voortkomen uit het vergeten van bepaalde dingen te tikken. Je kan een `pinMode` vergeten te tikken in de setup en je led zal niet branden. Je kan een `exitFlag` vergeten de tikken en je state-machine loopt vast.

Nu hebben we een gedeelte op kunnen lossen door middel van functies en door middel van macro's. In deel 3 leren we slechts hoe we de computer voor ons hele projecten met state-machine skeletten voor ons kunnen laten aanmaken. Door de computer dit te laten doen voor ons, kunnen wij nog minder dingen vergeten. Daarbij houden we de codestructuur en syntax aan die ik tot nu gebruikt heb.

## Bijlage 1 De software timers

Ik heb al eerder een hint gegeven over hoe de software timers werken. Een timer zoals ik ze gebruik decrementeert naar nul toe. Of in simpelere woorden. Ze tellen af naar nul. Misschien zit je met deze vraag: “waarom doe je zo moeilijk en gebruik je niet gewoon millis() zoals iedereen”.

Ik heb daar meerdere redenen voor. Om te beginnen is millis() een erg inefficiënte methode. millis() retourneert namelijk een 32 bits waarde. Je hebt ook een 32 bits waarde nodig om de vorige waarde in op te slaan. Een standaard millis implementatie ziet er zo uit.

```
#define interval 500
// of const int interval = 500;
unsigned long previousMillis = 0;

void loop() {

    if(millis() - previousMillis >= interval) {
        previousMillis = millis();

        // code
```

Dit voorbeeld kennen we allemaal. Maar hoe werkt het nu precies in de Arduino? millis() is een functie die een 32 bits waarde van een variabele retourneert, deze variabele wordt opgehoogd in een timer Interrupt Service Routine. Deze timer ISR wordt op de achtergrond uitgevoerd. Dit timer ISR waarin de millis counter staat, heeft Arduino voor ons opgezet op de achtergrond. We zien hem niet maar hij is er wel degelijk.

Als eerst worden er vier bytes afgetrokken van vier andere bytes. En het resultaat wordt vergeleken met nog eens vier bytes. Ik vind het persoonlijk zonde om vier hele bytes geheugen toe te wijzen aan een actie en telkens zoveel instructies moet doen om slechts een ledje te laten knipperen elke seconde. De atMega328P die op meeste Arduino's zit is een 8-bit micro controller. Dat houdt in dat hij met slechts een byte tegelijk kan rekenen.

Wat doe ik nu anders dan millis? Ik heb al veel deze constructie gebruikt:

```
#define interval 500
// of const int interval = 500;

void loop() {

    if(!timerT) { // zelfde als if(timerT == 0)
        timerT = interval;

        // code
```

Ik kijk of een byte al 0 is geworden en zo ja, dan zet ik er een nieuwe 8 bit waarde in.

Ik zal nu in detail uitleggen hoe het werkt en hoe dit verschilt met millis(). Ik gebruik per software timer een 8 bits globale variabele die in de timer ISR aftelt naar 0. Om dat te doen moeten we zelf een timer ISR aan de praat krijgen. Dit is voor beginners niet makkelijk om te doen. Gelukkig hebben erg veel mensen dit al voor ons uitgevogeld.

Ik gebruik hardware timer2 om een ISR elke ms uit te voeren. Met de functie initTimers() start ik het ISR op. Hoe dit precies werkt, valt buiten de scope van deze tutorial. Bovendien ben ik niet daartoe het meest geschikt voor. Ik laat slechts zien hoe we de taak gedaan krijgen.

Ik laat wel de ISR zien met zes software counter ingevuld. Dit is wel mijn creatie.

```
volatile unsigned char task1T, task2T, task3T, task4T, helloT, worldT;

ISR(TIMER2_COMPA_vect) {
    static unsigned char _1ms, _10ms, _100ms;

    // 1ms timers
    _1ms += 1;

    if(task4T) task4T--;

    // 10ms timers
    if(!(_1ms % 10)) { _1ms = 0; _10ms += 1;

        if(task3T) task3T--;
        if(helloT) helloT--;

    // 100ms timers
    if(!(_10ms % 10)) { _10ms = 0; _100ms += 1;

        if(task1T) task1T--;
        if(worldT) worldT--;

    //1000ms timers
    if(!(_100ms % 10)) { _100ms = 0;

        if(task2T) task2T--;

    }
    }
    }
}
```

In dit voorbeeld gebruik ik drie bytes aan geheugen om met verschillende tijdbases te werken. Als eerste hebben we de \_1ms. Deze wordt elke ms verhoogd. Daarna komt de eerste software timer, task4T, deze wordt met een if-statement bekeken of iets anders is dan nul. In dat geval wordt hij verlaagd met 1. Zo zorgt dit ISR er voor dat task4T elke ms wordt verlaagd totdat hij nul is. Zo kan ik dus een waarde in task4T stoppen in een entryState van een state-machine en in de onState kan ik wachten totdat hij 0 wordt.

```
if(!(_1ms % 10)) { _1ms = 0; _10ms += 1;
```

Deze if-statement zorgt er voor dat het volgende code blok elke 10ms wordt uitgevoerd i.p.v. elke 1ms. Als eerste: `_1ms % 10` is pas nul wanneer `_1ms` gelijk is aan 10. Bij elke waarde van `_1ms` die

geen veelvoud is van 10 is het resultaat van `_1ms % 10` niet nul. Daarom inverteren we het resultaat met `!` opdat de code achter de accolade { elke 10ms wordt uitgevoerd.

Wat gebeurt er nu als dat het geval is? Om te beginnen wordt `_1ms` op nul gezet zodat hij opnieuw kan optellen naar 10. Dan verhogen we de volgende software timer, `_10ms`.

Op deze wijze verdelen we het ISR in blokjes die elk een eigen tijdbasis hebben. We zijn niet verplicht om deze bases telkens te delen door tien. We kunnen ook een basis maken van een minuut of twintig secondes als je dat wilt.

Een van de voordelen van deze methode is dat we slechts een byte per software timer gebruiken plus drie extra bytes in het ISR voor de tijdbases. Het aftellen in plaats van optellen is fijn omdat er dan geen overflows plaatsvinden waarmee we rekening zouden moeten houden.

Een ander voordeel is dat dit:

```
if(millis() - previousMillis >= interval) {  
    previousMillis = millis();
```

simpelweg veel meer tikwerk is dan:

```
if(!timerT) {  
    timerT = interval;
```

Het laatste voordeel is dat het gebruik van deze timers ontzettend simpel is. Ze werken ook uitstekend in combinatie met onze state-machine structuur.

Er is een nadeel van deze methode ten opzichte van `millis()`. Als je een timer wilt toevoegen dan moet je op drie plaatsen iets aanpassen; de declaratie in `timers.cpp`, de decrementatie in het ISR en de externe declaratie in `timers.h` (wordt uitgelegd in deel 3). Dit probleem heb ik echter getackled op een aparte wijze.

Het mooiste van deze methode is dat `timers.cpp` en `timers.h` bij mij worden gegeneerd voor mij door een python script. Details over hoe dat precies werkt komt in deel 3 aan bot. Voor nu laat ik alleen zien hoe ik op het moment een timer toevoeg. Het volgende voorbeeld is een simpel tekst bestandje genaamd `timers.tab` (tab staat voor tabel)

```
task1T 100  
task2T 1000  
task3T 10  
task4T 1  
helloT 10  
worldT 100
```

Als ik een timer wil toevoegen, wil verwijderen of een tijdbasis wil aanpassen dan doe ik dat in dit bestandje. Dan dubbelklik ik op een bestand genaamd 'updateTimers.py' en `timers.cpp` en `timers.h` worden opnieuw aangemaakt met de wijziging. Het generen van code is ook krachtige tool om bugs te voorkomen en om sneller te kunnen werken.

Het voordeel is dat ik op deze wijze nog maar 1 plaats in 1 bestand hoeft aan te passen. Ik gebruik hetzelfde truukje om IO aan te maken. Ook dat staat haarfijn uitgelegd in deel 3.