Practicum handleiding

INLEIDING COMPUTERGEBRUIK

DEEL 3

Programmeren

in Java

V7.2 Sept 2007

Te doen *ruim* voor je aan het practicum begint:

- **Leer Hoofdstuk 1**
- **Doe de blackboard toets voor ICG3**
- **Blader Hoofdstuk 2 door**
- **Gebruik Hoofdstuk 2 op het practicum zelf**
- **Kom op tijd = 13:30 !**

NB: De laatste versie van alle documentatie, dus ook deze handleiding staat op blackboard. Die laatste versie wordt gebruikt voor het practicum.

Inhoudsopgave

Hoofdstuk 1

Programmeertalen

1.1 Inleiding

Een werkend programma in een computer is niets anders dan een lange lijst van machinegecodeerde binaire instructies. Iedere instructie bereikt op zich slechts weinig, waardoor een klein programma al uit duizenden instructies bestaat. Bij de eerste computers werden deze instructies met de hand (met wipschakelaartjes!) in het werkgeheugen gezet. Dit is geen praktische manier van werken. Bovendien heeft ieder type processor (Intel Pentium, Motorola PowerPC, …) zijn eigen verzameling instructies (*instructieset*), waardoor een dergelijk programma niet overgebracht kan worden van de ene computer naar de andere. Om het probleem van de machineafhankelijkheid te omzeilen, werden machineonafhankelijke *hogere programmeertalen* ontwikkeld. Een programma is nu een reeks *opdrachten* (statements) naar een programmeertaal. Een dergelijk programma wordt door een zogenaamde compiler omgezet in machinecode. Verschillende computers hebben verschillende compilers, waardoor hetzelfde programma in een hogere programmeertaal wordt gecompileerd tot de correcte machinecode voor machines van verschillend fabrikaat (Intel, Motorola, HP, IBM,…).

1.2 Het compilatieproces

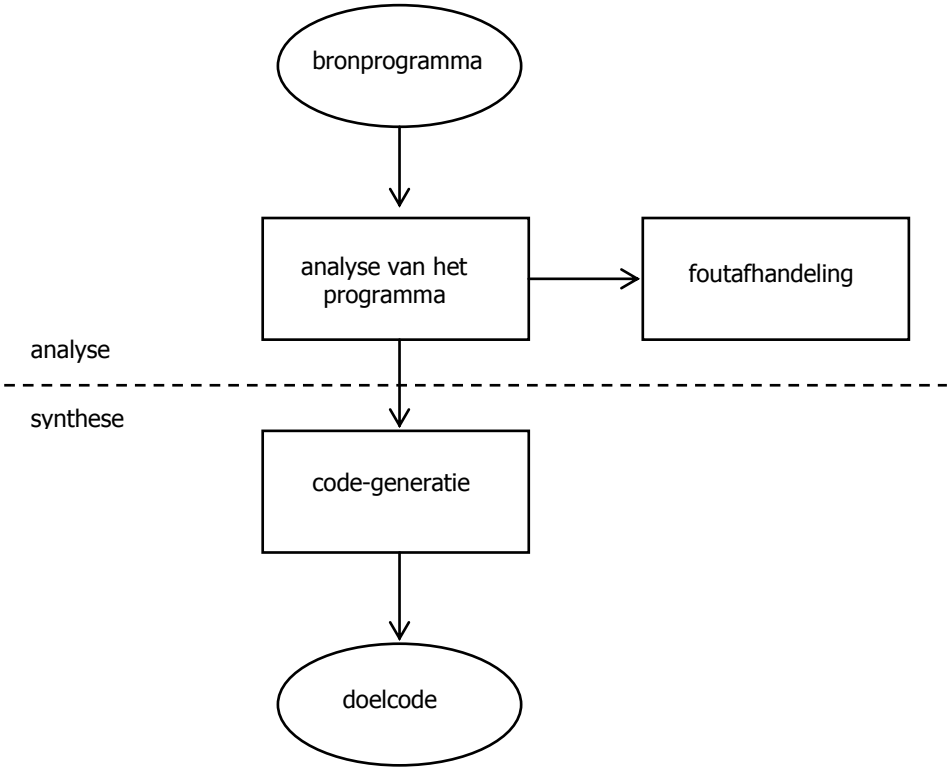
Terminologie
Een *vertaler* (compiler) is een programma dat een *bronprogramma* (source program), geschreven in een *brontaal* (source language), vertaalt naar een equivalent *doelprogramma* (object program) geschreven in een *doeltaal* (object language). De brontaal zal in het algemeen een programmeertaal van hoog niveau zijn (C, C++, Java,…). De doeltaal zal meestal een taal van laag niveau zijn (machinecode / assembler). Overigens kan elke taal als doeltaal optreden; men kan bijv. een vertaler van C++ naar C realiseren of van een wiskundige taal zoals Maple / MatLab naar C.

Een *interpretator* (interpreter - spreek uit inteurprutter) voor een brontaal is een programma dat programma's in die brontaal meteen uitvoert. Het significante verschil tussen een interpreter en een compiler is dat een interpreter de sourcecode direct interpreteert en uitvoert, terwijl een compiler het programma eerst omzet in objectcode die later kan worden uitgevoerd. Hoewel interpreters heel gemakkelijk in het gebruik zijn, kost dit redelijk wat overhead. Een interpreter kan wel 20 keer zoveel processortijd kosten als een gecompileerd programma. Een interpreter kan beschouwd worden als een compiler die per ingetypte regel vertaalt en dus ook per regel foutmeldingen geeft. Hoewel er C intepreters zijn gemaakt, is het vaak niet handig om een taal die voor compilers is ontworpen van een interpreter te voorzien. Bij een interpretatieve taal wil je zo weinig mogelijk afhankelijkheid tussen de ingevoerde regels. Vandaar dat algebraïsche talen zoals Maple, Matlab en Mathematica zich goed lenen voor interpretatie. Interpreters zijn echter al vrij oud; oude generatie interpreters zijn APL en Basic. Een redelijk nieuwe interpreter is Java. Dit is speciaal door SUN ontwikkeld om met webpagina's complexere dingen te doen. In een webpagina op een web server staan regels Java source code, die op de PC / Mac / Linux systeem van diegene die de webpagina bekijkt wordt geïnterpreteerd (als Java goed geïnstalleerd is op zijn machine). Java kan (zoals veel interpreters) ook gecompileerd worden.

Overzicht van het compilatieproces
Hoewel er grote verschillen in de feitelijke opbouw van een compiler kunnen bestaan, is een aantal functionele componenten steeds terug te vinden. Zo vindt binnen een compiler altijd eerst een complete *analyse* van de sourcecode (het bronprogramma) plaats, waarna een *synthese* van de

objectcode (doelprogramma) volgt. Een schematisch overzicht van het gehele vertaalproces wordt in figuur 1.1 gegeven.
In de analysefase wordt de sourcecode geanalyseerd en worden alle fouten die eventueel in de sourcetext aanwezig zijn gemeld. Indien er geen fouten zijn wordt de sourcetext vastgelegd in een tussenvorm (*intermediate representation*). Dit wordt gedaan om de compiler op te splitsen in twee delen. Een deel dat onafhankelijk is en een deel dat afhankelijk is van het type processor waar je naar toe wilt vertalen. Hierdoor kan zowel de analyse van de sourcetext als wel de codegeneratie worden geoptimaliseerd, omdat nu beide onafhankelijk zijn van de vernieuwing op het andere front (taaldefinitie en processorontwerp). De codegenerator genereert zgn. *object code* (doelcode). Deze object code is specifiek voor de architectuur van de machine (bijv. Intel Pentium) en is dus niet geschikt voor een andere architectuur (bijv. Mac / PowerPC).
Een belangrijk onderdeel van een compiler - *foutafhandeling* - zorgt er voor dat begrijpelijke foutmeldingen worden gegenereerd.

Het vertaalproces
De programmatekst wordt meestal gemaakt met een gewone tekst-editor, bijvoorbeeld `vi` of `emacs` onder UNIX, maar met het gemak van de X-Windows omgeving ook met een desktop georiënteerde editor, zelfs met WORD/Open Office (in text-mode). Deze programmatekst wordt ingevoerd in de *compiler*, een vertaalprogramma dat deze opdrachten omzet in machine-instructies. Een opdracht in een hogere programmeertaal kan vertaald worden in vele duizenden machine-instructies. Deze instructies worden naar een bestand geschreven. Dit bestand kan vervolgens in het geheugen worden geladen door de *loader* en worden uitgevoerd (*execution*).



Figuur 1.1. Structurering van een vertaler

Het hierboven beschrevene is correct voor kleine programma's die alleen eenvoudige berekeningen uitvoeren en geen in- en uitvoer doen. Voor grote programma's is er nog een tussenstap noodzakelijk. Eenvoudige wiskundige bewerkingen zoals optellen en vermenigvuldigen worden door de compiler vertaald naar machine-instructies voor vermenigvuldigen en optellen, maar voor het berekenen van een wortel of een logaritme bestaat vaak geen enkelvoudige machine-instructie. Voor de uitvoering van zulke functies zijn grote aantallen machine-instructies nodig. Deze berekeningen zijn beschikbaar in al vertaalde bibliotheken in de vorm van standaard routines. Er bestaan ook zulke bibliotheken voor in- en uitvoer. De benodigde bibliotheekroutines worden door een ander programma, de *link editor*

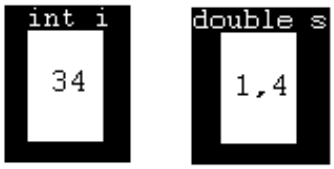
(linker) aan de gecompileerde code van ons programma (*object module*) toegevoegd tot het complete bestand: *executable code*. Deze code kan in het geheugen worden geladen, waarna de besturing aan het programma wordt overgedragen.

1.3 Java

Variabelen

Programmeertalen werken veel met variabelen. Een variabele kun je je voorstellen als een vakje met een naam en een waarde. De naam van de variabele kan een heel woord zijn, maar vaak worden enkel letters gebruikt. Een variabele heeft ook een type, bijvoorbeeld *int* (van *integer* – spreek uit intudsjur), *double* of *boolean* (*true* of *false*). Dit type moet altijd worden gedeclareerd voordat er met de variabele wordt gewerkt. In Java gebeurt dit door te typen:

```
int i = 34;
double s = 1.4;
```



Er zijn nu twee variabelen gemaakt, een `int` met de naam `i` en waarde `34`, en een `double` met de naam `s` en de waarde `1.4`. Een *integer* kan alleen maar gehele getallen als waarde hebben. Een *double* is een getal die ook gebroken waarden kan hebben. Een double moet altijd in de engelse notatie geschreven worden, dus met een punt!

If Then Else statement

Om een programma in bepaalde gevallen iets wel of juist niet te doen bestaan er *if statements*. Het werkt als volgt: de code tussen de haakjes achter `if` moeten `true` zijn om de code na de `if` tussen de accolades uit te laten voeren. Dit heet de `Then` clause. `True` en `false` betekenen 'waar' en 'niet waar'. Bijvoorbeeld:

`(1 < 2)` is `true`, maar `(2 < 1)` is `false`. Als het *if statement* `false` is wordt de code overgeslagen. Als er na deze code nog een `else` staat wordt de code tussen de accolades na de `else` uitgevoerd als het *if statement* `false` is. Een voorbeeld van een *if then else* statement

```
int i = 3;
if(2 == i) {
   System.out.println("variabele i heeft de waarde 2");
}
else {
   System.out.println("variabele i heeft een andere waarde dan 2");
}
```

De code geeft als output "variabele i heeft een andere waarde dan 2"
Onthoud dat in een programmeertaal geldt dat `a = 5` betekent dat `a` de waarde `5` krijgt en dat `a == 5` een `true` teruggeeft als `a` gelijk is aan `5`.

Loops

Om een bepaald stuk code meerdere malen uit te laten voeren, bijvoorbeeld tien keer, ben je niet verplicht om de code tien maal in je bestand te zetten. Je kunt dan gebruik maken van *loops*, het houdt in dat alles wat tussen de accolades van de *loop* staat, herhaald wordt totdat er aan een bepaalde voorwaarde voldaan wordt. Om een stukje code tien keer te herhalen, moet je eerst een teller aanmaken en deze elke keer op te hogen wanneer de code is uitgevoerd en te stoppen bij 10.
Dat gebeurt als volgt:

```
for (int i = 0; i < 10; i++) {
        …
}
```

Dit wordt een *for-loop* genoemd. Zoals je ziet bevat hij drie parameters, met `int i = 0` wordt er dus een nieuwe variabele gemaakt, `i < 10` betekent in een for-loop 'zolang `i` kleiner dan 10, blijf loopen' en `i++` wil zeggen dat elke keer als de loop doorlopen is, verhoog `i` met 1. In de eerste opdracht van het practicum staat zo'n for-loop in de java code. Probeer een ingewikkelde for-loop altijd stapsgewijs te analyseren.

Switch
Om code uit te voeren afhankelijk van de waarde van een variabele kunnen we *if* statements gebruiken:

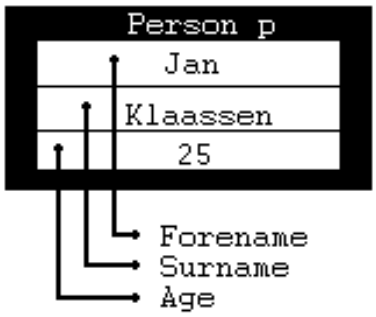
```
if (i==0)
{
  // code voor 1
}
else
   if (i==1)
   {
      // code voor 2
   }  etc.
```

Deze constructie is vaak wat omslachtig en kan vervangen worden door het *switch* statement:

```
switch(i)
{
   case 0:
      // code voor 1
   break;
   case 1:
      // code voor 2
      break;
   default:
      // code als i niet een waarde heeft uit de bovenstaande gevallen (hier 0 of 1)
   break;
}
```

Deze methode maakt je code meestal wat overzichtelijker, maar de keus is aan jezelf. Vergeet het *break* statement niet, want dat geeft aan dat de rest van de code binnen de switch niet uitgevoerd moet worden.

Objecten
Een object kun je zien als een voorwerp die verschillende eigenschappen bevat. Bijvoorbeeld het object Person. De eigenschappen van deze Person zijn dan voornaam, achternaam en leeftijd (resp. Forename, Surname en Age). Deze eigenschappen horen dan bij het object Person. Het aanmaken van een object Person gebeurt als volgt:




```
Person p = new Person;
```

Het toewijzen (of veranderen) of opvragen van de gegevens van het object, gebeurt met de zgn. get- en setmethoden. Dus:

```
p.setForename("Jan");
p.setSurname("Klaassen");
p.setAge("25");

p.getAge();  // geeft terug -> 25
```

Toegepast op de Jobot, daar worden voor de sensoren ook getmethodes gebruikt; `getJobot().getSensor(0);` Deze geven waardes terug tussen de 0 en 1023.

Functies
Vaak is het handig om een compleet stukje code in zijn geheel vanuit verschillende programma onderdelen aan te kunnen roepen. In de class `JobotBaseController` zijn een aantal handige functies opgenomen, die in de code wordt aangeroepen. Bijvoorbeeld `getJobot()` is een functie die uitzoekt waar de definities van de robot staan en die teruggeeft. `getSensor(x)` haalt de waarde van sensor X op en geeft die waarde terug. Een functie is dus een stukje code dat je als complete eenheid aanroept en dat je een parameter meegeeft en dat een waarde terug kan geven.

Soms zijn er functies die alleen maar een handeling verrichten, bijvoorbeeld `drive(a, b, c)` stuurt alle drie de motoren aan. Hiervoor hoeven we geen waarde terug te krijgen. Als een functie niets teruggeeft, wordt er bij die functie het woord `void` toegevoegd. Void betekent leeg, dus wordt er niets (een leeg veld) teruggegeven.

Hoofdstuk 2

Het in Java programmeren van een robotje

2.1 Inleiding

In deze derde dag van het ICG practicum is het doel om je kort te laten zien wat programmeren inhoudt. We laten je een robotje programmeren dat iets waarneemt met sensoren, berekent, en dan iets uitvoert (laat bewegen). In robot jargon een "Sense–Reason-Act" cyclus. Het programmeren van het omnidirectionele robotje gaat eerst met behulp van een simulator. In tegenstelling tot het vorige practicum onderdeel werken we nu weer onder Windows-XP. Eerst wordt een bestaande simulator geladen, waaraan eigen stukken Java code kunnen worden toegevoegd. De Java code die moet worden gemaakt blijft vrij basaal, maar genoeg om een indruk te krijgen van de taal en van het programmeren als activiteit. Nadat de simulator laat zien dat je robot doet wat je wilt, kun je de software downloaden op een echt robotje en kijken wat hij doet.
Tijdens dit practicum is het de bedoeling dat je een verslagje bijhoudt. Er wordt in de handleiding aangegeven wanneer er van je verwacht wordt iets in het verslagje te zetten.
Mocht je iets over de joBot niet helemaal begrijpen, zoek het dan op in de documentatie! (zwarte map, of de pdf op blackboard)

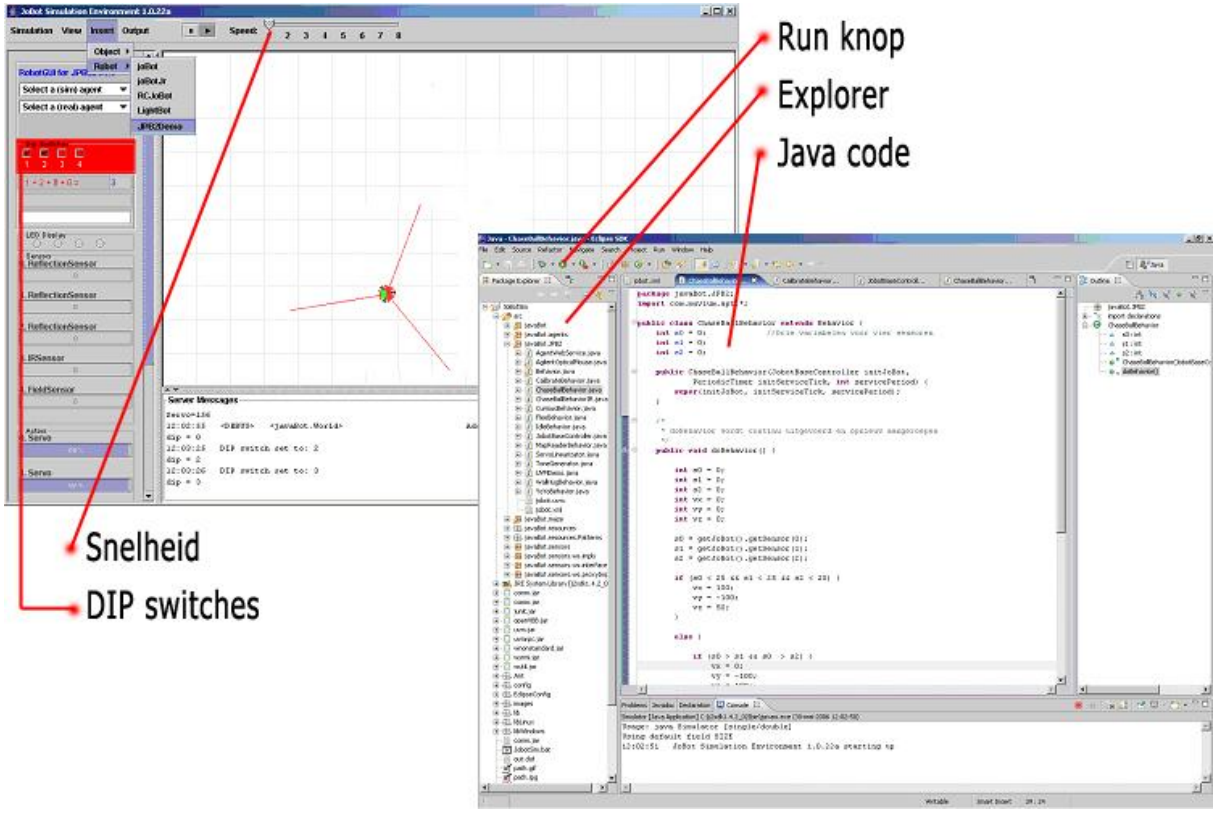
2.2 Werkvoorbereiding

- Je werkt deze dag in je eentje.
- Log in op het netwerk onder je eigen studenten account
- Eerst wat huishoudelijk werk:
  - Ga via de Windows verkenner, of 'My computer' op de desktop naar de i:\ schijf. Ga vervolgens naar de map: i:\TNW-Practica\TN1402. Hier staat een batchfile genaamd `copy_to_h. Als` je op dat bestand dubbel klikt, wordt alle benodigde software op je eigen schijf gezet. Hierin wordt alles wat je doet in opgeslagen. Mocht je onverhoopt helemaal opnieuw willen beginnen, dubbelklik dan opnieuw op het bestand copy_to_h.
  - Klik alle windows dicht

2.3 Inwerken op ICGDemo

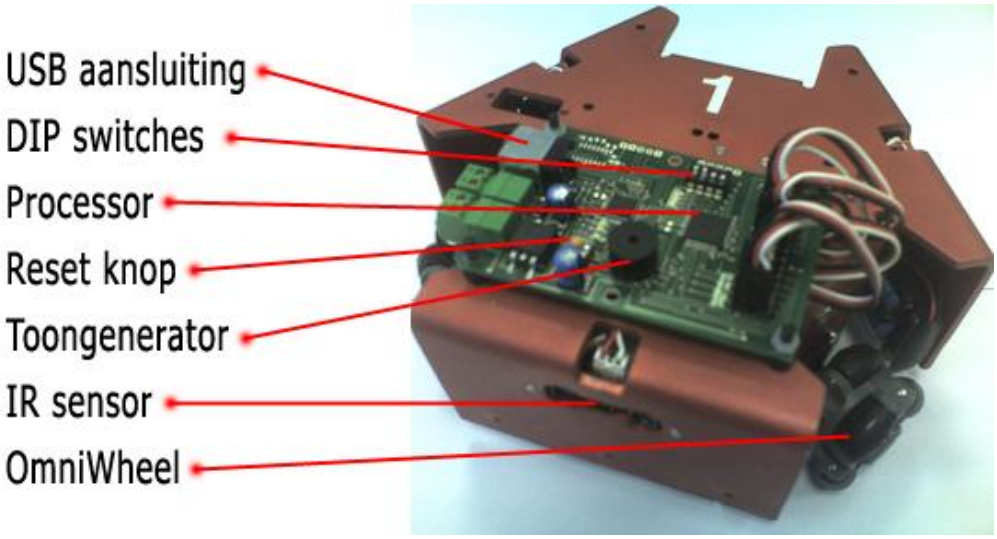
- Dubbel klik op het programma (de snelkoppeling) `eclipse` op je desktop.
- Eclipse start nu op, met het programma JobotSimICG alvast geladen
- Klik nu op de groene > button (run: 5de van links).
- Je krijgt nu een window met de JoBot Simulator, die je zojuist gestart hebt (zie ook afbeelding op de volgende pagina)
- Selecteer in het simulator menu `Insert/Robot/ICGDemo`. Je laadt dan het ICGDemo programmaatje in de simulator. Je ziet een robot met in rode lijnen het bereik van zijn afstandssensoren. Je kunt hier een beetje mee spelen. Haal bijvoorbeeld een bal op met `Insert/Object/Ball` en verschuif die met de muis. Ook de robot kun je zo bewegen.
- Links in het beeld is er een `RobotGUI` (Graphical User Interface) bij gekomen, gebruik die om met het robotje en de bal te spelen. Je kunt daar ook de 3 sensor waarden aflezen.

- De verschillende bewegingsprogramma's (behaviors) van de robot kun je selecteren met de ***dipswitches***. In de documentatie uVMDemo kun je vinden welke waardes voor de DIP switches overeen komen met de behaviors.



2.4 Inwerken op de JoBot

- Gebruik hiervoor de handleidingen van UVMDemo (dit is hetzelfde als de ICGDemo) en de joBot en pak een robot
- De joBot heeft de volgende onderdelen:



Aan/Uit schakelaar

- De joBot werkt met oplaadbare batterijen die ongeveer 30 minuten meegaan. Wees zuinig met de batterijen. De aan/uit schakelaar van joBot zit aan de rechterkant van het processorbordje en heeft drie standen:
  1. De middelste stand is UIT. *Zorg dat je robotje altijd uit staat als je hem niet gebruikt*.
  2. De stand naar het processorbordje toe is de programmeerstand. Hierbij staan alleen de processor en de sensoren aan, de motoren staan dan uit.
  3. De stand van het bordje af is de stand om het robotje te laten rijden.

Batterij

- De joBot heeft twee sets oplaadbare batterijen. Kijk onderin! In het midden onderin zit een 9 volt batterij die de processor en de sensoren voedt. Aan de zijkanten zitten twee batterijhouders voor 1.2v batterijen, die de motortjes voeden. Deze batterijen zijn groter omdat de motoren veel meer stroom verbruiken. Beide batterijpacks gaan ongeveer een half uur mee.

DIP Switch

- Met de DIP (Dual In-line Package) switch kun je het robot gedrag (behavior) kiezen. De schakelaar heeft 4 aan/uit schuifjes. Naar boven is aan, naar onderen uit. Een switch heeft van links naar rechts de waarde: 2^0, 2^1, 2^2, 2^3, ofwel 1, 2, 4 en 8. Door de waardes op te tellen krijg je combinaties tussen 0 en 15. Kijk in de UVMDemo (ICGDemo met een wat andere naam) documentatie welk behavior met welke stand overeenkomt. In de Grafische User Interface (GUI) van de simulator kun je dezelfde switches ook vinden.

Processor

- De robot is uitgerust met een kleine 40 MHz computer met 128 K geheugen. Dit computertje bestuurt alle elektronica en kan java programma's uitvoeren. De meeste java implementaties werken met een Java Virtuele Machine (JVM), een interpreter die langzaam is. In joBot wordt gebruik gemaakt van de muVium compiler die de java code compileert naar machinetaal, waardoor de code snel executeert en compact is. Deze compiler wodt aangeroepen zodra je de software in het robotje (de joBot) gaat downloaden.

OmniWheel en servomotors

- De joBot maakt gebruik van drie servomotortjes, zoals die in modelbouwautootjes worden gebruikt. Ze hebben omnidirectionele wielen die het mogelijk maken om zonder veel wrijving zijwaarts te bewegen. Hierdoor kan het robotje in iedere richting rijden zonder zijn oriëntatie te hoeven veranderen.

Connectors

- De connectors zijn aangesloten op de bus van de processor. Als er eentje losraakt vraag dan aan de assistent om dit te herstellen. Het verkeerd aansluiten beschadigt de sensoren!

IR-Sensors

- De sensoren zenden een dunne straal infrarood licht uit, de rechterkant is de zender, de linkerkant de ontvanger. Licht dat weerkaatst in de ontvanger wordt gebruikt om via de hoek te berekenen wat de afstand is. Hierdoor is de sensor niet afhankelijk van de kleur van het object dat het licht terugkaatst. Met de sensoren kan vrij nauwkeurig de afstand tot een object gemeten worden. De waarden die je terugkrijgt van de sensoren lopen overigens in het programma van 0-1023, hoe hoger hoe dichterbij. Je gaat zelf het betrouwbare bereik van de sensor bepalen.

Reset

- De reset knop wordt gebruikt om het programma te herstarten als het mocht vastlopen. De knop wordt ook gebruikt bij het programmeren van de robot om de processor in de "bootmode" te zetten, waarin hij via de USB kabel geprogrammeerd kan worden.

USB aansluiting

- Deze connector wordt met een kabeltje aangesloten op de USB poort van je PC. Let op: de communicatie programma's zijn zo ingesteld dat je de linker usb poort aan de voorkant van de computer moet gebruiken.

Het ICGDemo programma

- Het programma dat in het robotje draait en zijn "brein" vormt, noemen we ook wel een agent. Het ICGDemo programma is al in de joBot geladen, maar....
- *De vorige gebruiker kan junkcode op de robot achtergelaten hebben. Dus voordat je iets in het robotprogramma wijzigt moet je de joBot eerst weer met de originele ICGDemo laden.*
- Sluit daarom de robot op de PC aan met de USB kabel. Zet de robot aan in de programmeermode, met de schakelaar naar het processor bordje toe. De processor en de sensoren werken, maar de motoren niet. **Als de schakelaar op UIT staat, tijdens het opstarten van uvmide.exe zal het programma de USB poort niet herkennen. Als de batterijspanning van de 9v batterij onder de 7.5 volt daalt zal de USB poort ook niet meer goed werken.**
- Dubbel klik op de desktop op de snelkoppeling *UVMIDE*. Het hulpprogramma *uVM-IDE* waarmee je de software op de robot kan uploaden start nu op.
- Open `File\H:\Apps\ICG\Workspace\JobotSimICG\Jobot29ICG.xml (staat al in het lijstje in het File menu!)`, in dit bestand staat o.a. welke bestanden er allemaal naar de robot worden geupload.
- Selecteer `Project\Package`. De code wordt nu gecompileerd. Wacht op DONE in het zwarte *UVMIDE* windowtje. Na DONE komt de tijduur te staan, ongeveer 6 seconden. Package moet je in het vervolg gebruiken als je de programma code van het robotje veranderd hebt.
- Selecteer `Project\Connect and Reboot`
- Duw nu binnen 10 seconden op de reset knop op het robotje. Onderin komt in de status regel 'Boot mode OK!' te staan. Zo niet, dan was je te laat, en probeer het nogmaals.
- Selecteer nu `Project\Upload`
- In het zwarte venster komt eerst te staan dat het geheugen gewist (erased) is. Nu begint het eigenlijke uploaden, wat een minuut kan duren. Wacht nu tot je upload succesvol ziet onderaan het *uVM-IDE* window. Links van de statusregel wordt er een balk langzaam gevuld van links naar recht, dit geeft de voortgang aan. Zie je geen voortgang, meld dit dan even.
- Als het uploaden klaar is. Selecteer dan Project\Run, om de software te starten.
- Sluit dan het *uVM-IDE* venster, en zet **daarna** het robotje uit.
- Bovenstaande procedure moet je telkens weer volgen als je de programma code hebt aangepast.

- Bij het aanzetten van de robot (schakelaartje van het processorbordje af) start ICGDemo op in de "run" mode. Eerst wordt door de processor gekeken wat de verschillende hardware onderdelen zijn zoals servo's, sensoren en leds. Daarna wordt o.a. een "heartbeat" timer gestart die vanaf dat moment het gedrag van de joBot dirigeert. Iedere heartbeat kijkt het programma wat er gedaan moet worden. Wat er tijdens een heartbeat wordt gedaan is afhankelijk van de "state" van de agent. In de ICGdemo wordt deze bepaald door de stand van de DIP switches. Als ICGdemo ziet dat de stand van de DIP switches is gewijzigd, wordt de oude behavior gestopt en wordt er een nieuw behavior gestart. Het heartbeat lampje (de rode led) gaat aan en uit op het ritme van de heartbeat en geeft aan dat het programma nog draait. Als het niet meer knippert, weet je dat je programma in de fout is gegaan. In dat geval kun je de reset knop gebruiken.
- Als je de kabel eraan hebt laten zitten, kun je de info van het robotje in de *uVM-IDE* window bekijken: start het *uVM-IDE* programma opnieuw, laad de juiste file, klik op connect/reboot, en druk op de reset knop van de robot. Na een tijdje komt de debug informatie op het scherm. Als het scherm irritant gaat flikkeren, kun je op Alt+Tab drukken. En daarna nogmaals Alt-Tab om *uVM-IDE* weer te selecteren.
- Meestal is het beter om de kabel eraf te halen en de robot in de bak of gang te testen. (wel eerst het uvm-ide programma sluiten voor je de robot uit zet of de kabel losmaakt!)

2.5 SENSE opdracht

- Voordat je nu met de software aan de slag gaat, ga je eerst beginnen met een kalibratie van de IR-sensoren. In de theorie van paragraaf 2.4 heb je gelezen hoe de IR-sensoren gebruik maken van triangulatie en nu ga je aan de sensoren meten. De waarden die de sensoren terug geven zijn namelijk niet lineair verbonden met de afstand tot de sensor.
- *De opdracht is om de sensorwaarde als functie van de afstand uit te zetten en een linearisering uit te voeren om een zo goed mogelijke benadering te krijgen van de afstand.*
- Je hebt zojuist de software voor de robot opnieuw geüpload, als je dit niet heb gedaan moet je dit nu alsnog doen m.b.v. paragraaf 2.4.
- De robot moet aangesloten zijn op de computer, de schakelaar naar het processorbordje toe (op Program mode) en de DIP=1.
- Start *uVM-IDE* weer op.
- Klik `File/1` om het juiste xml bestand te laden
- Selecteer `Project/Connect and Reboot`, druk op de reset knop van de robot en wacht op 'Bootmode OK!'
- Selecteer `Project/Run`.
  Om de seconde begint het robotje de waarde van sensor S_0 (de sensor vlak tegen het processorbordje) naar de computer te sturen. Maak nu een tabel met een afstand van 0 tot 80 cm en de bijbehorende sensorwaarden. Voor een nauwkeurige meting, moet je ervoor zorgen dat je méér meetpunten pakt naarmate de waardes sterker gaan variëren. Als de sensor metingen bij dezelfde afstand veel varieren is de sensor misschien stuk, of zijn de batterijen op.
  Als het programma na een tijd problemen vertoont, gebruik dan de reset knop op de robot, en/of start het UVMIDE programma opnieuw op.
- Zet je meetpunten uit in een grafiek in excel (origin mag ook, als je de grafiek maar kan kopieren naar je verslag in Word). Wat valt je op aan de meetpunten? Trek je conclusies hier uit; op welke afstanden is de sensor betrouwbaar? Teken daarna de linearisering die jou het beste lijkt (in welk gebied moet de formule nauwkeurig zijn vind je?) en stel de formule ervoor op (ongeveer, heel nauwkeurig hoeft het niet in ons geval). Voeg de grafiek met de gevonden rechte lijn en formule toe aan je verslag.
- Sla je verslag meteen op nadat je iets veranderd hebt. Het uitschakelen van de robot als die aangesloten is op de computer kan een crash veroorzaken. Ook het uittrekken van de usb kabel zou een crash kunnen veroorzaken. Dit probleem ligt volledig aan het robotje en/of de speciale software die we hebben geïnstalleerd.

2.6 Vergelijk Simulator en joBot

- Je gaat eerst de java code bekijken die het robotje (en de simulator) aansturen.
- Klik in eclipse links bij de `package explorer` tab op de + van `jobotSimICG`
- `JobotSimICG` expandeert nu. Klik op de + bij `src`
- `src` expandeert nu. Klik op de + bij `javaBot.ICG`
- `javaBot.ICG` expandeert nu. Je ziet nu de sources van o.a de behaviors van de robot, zoals FleeBehavior.java (DIP=5). Als je er 2x op klikt, verschijnt de programma code in de source window van de simulator, boven-rechts.
- Je ziet ongeveer als eerste staan:
  ```
  public class Fleebehavior extends Behavior {
  ```
  Dit geeft aan dat dit stukje programma de eigenschappen erft van een parent class `Behavior`
  Het eigenlijke werk begint na:
  ```
  public void doBehavior() {
  ```
  na deze regel wordt namelijk het specifieke gedrag van de betreffende behavior gedefineerd.

- Bekijk de java code van het `WallHugBehavior` (dip=8) en het `ChaseBallBehavior (dip=11)`. Probeer te begrijpen wat de verschillende gedeeltes van de code doen. Lees altijd het commentaar dat bij de code staat
- Test deze behaviors ook in de Simulator en met de robot. (Er zijn kleine ballen beschikbaar)
- Schrijf in je verslag wat de belangrijkste verschillen zijn die je opmerkt tussen de echte robot en de simulator. Probeer ook te beredeneren waardoor deze verschillen ontstaan en zet die ook in je verslag. De crux zit hem in het verschil tussen de papieren, gesimuleerde werkelijkheid en de fysieke werkelijkheid.

2.7 REASON opdracht

- Zoek nu uit hoe het FLEE behavior (DIP=5) werkt in de code.
- Schrijf op in je verslagje wat het flee behavior doet.
- *Het is de bedoeling dat je als eerste opdracht een nieuw behavior maakt, het CURIOUS behaviour, dat zorgt dat het robotje juist naar een obstakel toe rijdt. Heb je dat gedaan zorg er dan voor dat het robotje niet tegen objecten aanbotst maar altijd op een afstand van ongeveer 10-15 cm blijft.*
- We hebben alvast een CuriousBehavior.java bestand aangemaakt (DIP=4). De code hierin doet hetzelfde als de code in FleeBehavior, maar deze maakt gebruik van for-loops en een switch statement. Probeer ook die code te begrijpen.
- Pas de `CuriousBehavior` aan, zodat die het juiste gedrag vertoont
- Als je programma is gemaakt, `run` het in de simulator (klik op de groene knop in eclipse!).
- Het kan zijn dat je een foutje hebt gemaakt waardoor het programma niet weet wat je bedoelt. Deze problemen worden onderin het scherm van Eclipse gemeld in het tab-blad Problems. Errors moet je oplossen, warnings mag je laten staan (zijn er ongeveer 100)
- Als je er niet uit komt, vraag dan de assistent om hulp, want anders blijf je te lang proberen.
- Bij ernstige verminkingen van de code, ga uit eclipse, delete de workspace `H:\Apps\ICG\workspace` en haal opnieuw de code op (zie sectie 2.2 van deze handleiding.
- Wanneer de joBot in de simulator doet wat jij wilt, kun je hem op de echte robot uploaden.
- Let op, eerst de robot aansluiten en in programmeer mode zetten, alvorens het programma *uVM-IDE* te starten. En na het uploaden eerst het programma afsluiten, en dan pas de robot uit zetten of van de usb kabel halen. Vergeet ook niet `Project\Run te gebruiken`
- Voor het uitvoeren van de opdrachten kun je de vloer van het practicumlokaal of de gang worden gebruikt. Zet na gebruik de schakelaar op de middelste stand zodat er geen lampjes meer branden om batterijen te sparen.
- De simulator kan een groot deel van de acties van de echte robot simuleren, maar niet alles. Interacties tussen instructies in de processor of tijdsafhankelijke componenten geven neveneffecten en kunnen soms tot andere resultaten leiden. Wellicht heb je tijdens het testen van het programma op de robot al ontdekt waarin de verschillen zitten (yo physicists!).
- Als het behavior goed op de robot werkt, laat het dan zien aan de assistent. Plak het stukje javacode wat je zelf hebt aangepast in je verslagje. (Vergeet die niet op te slaan)
- Als het niet werkt, ga dan terug naar de simulator, en bedenk wat het probleem kan zijn.

2.8 ACT opdracht

- Laad in eclipse de `chaseBallBehavior.java`. Bekijk de code en probeer deze te begrijpen. Net zoals bij de FleeBehavior begint het echte werk pas na de regel:
  ```
  public void doBehavior() {
  ```
- Probeer in je simulator de chaseBallBehavior (DIP = 11) uit. Vergeet niet om ook een bal object toe te voegen aan het speelveld. Bekijk wat de joBot doet.
- De Jobot kan geen onderscheid maken tussen de muur en een bal. Het is vervelend dat de Jobot zo ver kan kijken omdat hij dan snel de muur aanziet voor een bal. Je wilt liever dat hij

nog iets langer doorrijdt op zoek naar de bal, want voor hetzelfde geld ligt de bal vlakbij de robot maar net niet in het zicht van een sensor. De bal rolt immers niet een meter ver weg.

- Wijzig de code zodat de visie wordt beperkt, schat zelf een goede beperking in (gebruik je calibratie grafiek!!). Bedenk eerst goed wat je wilt doen, voordat je de code gaat aanpassen. Test je aanpassingen in de simulator en pas de beperking eventueel aan. Hint: Tussen de eerste if statement en voor de else statement staat wat de joBot doet als alle drie de sensorwaardes lager zijn dan 10. Lees alle commentaren.
- Als je code goed werkt in de simulator, zet dan jouw aangepaste stukje code in je verslag.
- Als de JoBot niets in zicht heeft maakt hij een cirkel beweging. Een effectievere zoekbeweging is een spiraal beweging. Hij kijkt dan namelijk eerst dichtbij en gaat steeds verderop zoeken.
- Bekijk de code van de cirkel beweging, probeer goed te begrijpen hoe die werkt en probeer er een spiraal beweging van te maken. Een goed begrip van de cirkelbeweging leidt sneller tot een begrip van de spiraalbeweging. Als je er niet uitkomt, bedenk dan eerst hoe je de robot rechtuit moet laten rijden (Translatie). Dan hoe je de robot om zijn as moet laten draaien (Rotatie). En dan hoe je uit de translatie en rotatie een cirkelbeweging moet maken. Er is zelfs een simpele formule voor; schrijf in je verslag die formule op. Probeer cirkels met verschillende stralen te ontwerpen en dan uit te testen in de simulator.
- Analyseer nu een spiraalbeweging, uitgaande van de cirkelbeweging en bedenk dat er een tijdsafhankelijke factor mee moet spelen. Hint: Het chaseBallBehavior wordt elke 20ms opnieuw aangeroepen.
- Test je code eerst in de simulator en upload je nieuwe code op de robot als beschreven in §2.4. Kijk daarna of de robot in het echt ook werkt. Zet je aangepaste stukje code weer in het verslag. Laat de spiraal op de echte JoBot aan de assistent zien!
- Ben je snel klaar? Verzin dan een manier om te zorgen dat de robot de bal niet te hard wegstoot. Maar, zorg er wel voor dat hij er zo lang mogelijk snel op af gaat.

2.9 Verslag

- Download een conceptverslag van BlackBoard.
- Geef de file de naam icg3-<jouwnaam> en mail hem als je klaar bent naar: p.p.jonker@tudelft.nl met cc: peter@lithp.nl en subject: icgd3-<jouwnaam>. Vergeet niet het verslag te attachen. Bewaar zelf een kopie van je email als bewijs voor het opsturen.
- Controleer voor je hem opstuurt of je verslag voldoet aan de punten hieronder:

  Algemeen:
  -Namen, studienummers teamleden
  -Nummer van de robot en eventuele problemen die zijn ontstaan

  Samenvatting:
  -Korte globale beschrijving van het robot practicum. Wat heb je gedaan

  Sense opdracht / Calibratie:
  -Sensorwaarde tegen afstand uitgezet in een tabel en in een grafiek
  -Linearisatie
  -Interval van betrouwbaarheid

  Vergelijk Simulator en joBot:
  -Verschillen tussen simulator en de joBot van drie behaviours
  -Beredenering van de verschillen

  Reason Opdracht:
  -Beschijving van het `FleeBehavior`
  -Stukje aangepaste javacode om `CuriousBehavior` te maken

Act Opdracht:
-Stukje aangepaste javacode om sensorbereik in te korten
-De formule en de waardes van twee verschillende stralen
-Stukje aangepaste javacode voor de spiraalbeweging van de joBot

Conclusies en bevindingen:
-Technische conclusie
- Persoonlijke impressie van het robot practicum

2.10 Tenslotte

- Ruim de boel op, zet je robotje en PC uit en hang de robot aan de lader in het kastje.

That's all Folks!