HOGESCHOOL ROTTERDAM / CMI

Practicumhandleiding Besturingssystemen

Onderdeel ArduinOS

TINBES03

Cursusjaar: 2022–2023
Auteur: dr. W. M. Bergmann Tiest
Met dank aan ir. J. de Hooge en dr. ir. M. Hajian.

Inhoudsopgave

1 Inleiding

In dit deel van de cursus Besturingssystemen ga je zelf een besturingssysteem bouwen voor de Arduino Uno: **ArduinOS**. De Arduino kan via een IDE in C geprogrammeerd worden, maar kan op die manier maar één programma tegelijk uitvoeren. Om een ander programma te starten moet hij opnieuw geflasht worden. Ook beschikt hij niet over een bestandssysteem. In deze opdracht worden deze problemen opgelost door een besturingssysteem te bouwen dat meerdere programma's tegelijk kan uitvoeren (multitasking), en een bestandssysteem heeft waarin bestanden opgeslagen kunnen worden, ook als de voeding wegvalt. Het besturingssysteem voert programma's uit die bestaan uit *bytecode*, beschreven in appendix A. Om aan deze bytecode te komen, moet bijvoorbeeld een C-programma gecompileerd worden met een daarvoor geschikte compiler. Het maken van zo'n compiler valt buiten de scope van deze opdracht, maar kan als extra worden gedaan. Voor deze opdracht worden enkele voorbeeldprogramma's in bytecode beschikbaar gesteld. Deze kunnen met behulp van het beschikbaar gestelde programma `convert` van de host-computer naar de Arduino gestuurd worden; zie appendix B.

2 Requirements

Het besturingssysteem

- kan commando's inlezen vanaf de *command line* via de seriële terminal.
- kan programma's in de voorgeschreven bytecode (opgeslagen als bestanden in het bestandssysteem) uitvoeren op de Arduino Uno of Nano.
- beheert een geheugen van tenminste 256 bytes.
- kan tenminste 25 variabelen van het type `CHAR` (1 byte), `INT` (2 bytes), `FLOAT` (4 bytes) of `STRING` (zero-terminated, variabel aantal bytes) in het geheugen houden, waarvan de waarde gezet, gelezen en gemuteerd kan worden.
- beheert een bestandssysteem ter grootte van het beschikbare EEPROM-geheugen.
- kan hierin tenminste 10 bestanden opslaan, teruglezen en wissen met bestandsnamen van maximaal 12 tekens (inclusief terminating zero).
- kan de nog beschikbare hoeveelheid opslagruimte weergeven.
- kan tot 10 verschillende processen bijhouden die gestart, gepauzeerd, hervat en beëindigd kunnen worden.
- houdt bij van alle variabelen bij welk proces ze horen, en geeft het geheugen dat de variabelen innemen vrij als het proces stopt.
- kan per proces 1 bestand tegelijk lezen of schrijven.
- houdt per proces een stack bij van tenminste 32 bytes.

In de volgende 7 paragrafen worden tips gegeven voor de implementatie van het OS. Je bent vrij om het op een andere manier te doen, als je je maar aan bovenstaande requirements houdt, je programmacode aan de code-standaarden (TI 3×3) voldoet en je de 7 checkpoints kunt laten zien. In appendix A wordt uitleg gegeven over de te gebruiken bytecode-taal.

3 Checkpoint 1: basisstructuur en ontwerp

Voor een groot softwareproject als dit is het belangrijk dat je van tevoren goed nadenkt over de structuur, zodat je niet op een later moment veel moet gaan herschrijven. Het OS bevat de volgende componenten:

Een command line interface waarin je commando's kunt typen en die herkend worden, zodat de bijbehorende code (functie) uitgevoerd kan worden.

Een bestandssysteem waarin je met een *file allocation table* (FAT) bijhoudt waar in het EEPROM files staan, hoe ze heten en hoe groot ze zijn. De FAT zelf staat ook in het EEPROM. Er moeten mogelijkheden

(functies) zijn om files toe te voegen, weer op te vragen, te wissen, en om de beschikbare vrije ruimte te zien.

Geheugenbeheer waarmee variabelen van de vier verschillende typen opgeslagen (van de stack naar het geheugen), weer opgevraagd (van het geheugen naar de stack), en gewist kunnen worden. In een *memory table* wordt van iedere variabele de naam en het type bijgehouden, bij welk proces hij hoort, en het adres en de grootte in het geheugen.

Procesbeheer waarmee je een nieuw proces kan starten en een gestart proces kan pauzeren, weer hervatten of afbreken. Van ieder proces wordt in de *process table* bijgehouden wat de naam van het programma is, het ID-nummer, de status, de *stack pointer* SP, de *program counter* PC, de *file pointer* FP en het adres van het begin van de loop. Verder moet ieder proces een eigen stack hebben. Ook moet er een functie zijn die van elk actief proces een instructie laat uitvoeren.

Voor ieder proces een stack waarop losse bytes gepusht en gepopt kunnen worden. Er moeten functies zijn om data van de vier verschillende typen de pushen en te poppen. Veel instructies kunnen argumenten van verschillende typen gebruiken. Daarom is het ook handig om een functie te hebben die een getal van een willekeurig numeriek type (`CHAR`, `INT` of `FLOAT`) kan poppen, en dit als een float teruggeeft.

Een manier om instructies uit te voeren die een byte van een bytecode-programma leest en de bijbehorende functie uitvoert.

Maak een ontwerp van de structuur en een lijst van benodigde functies met duidelijke functienamen en in commentaar een beschrijving van de argumenten, de return value en wat de functie moet doen. Laat dit door de docent checken.

4 Checkpoint 2: command line interface

Met een command line interface communiceer je met het OS door tekstcommando's te typen in een *console* bij een *prompt*, en krijg je antwoord ook als tekst op de console. Binnen de Arduino IDE kun je de Serial Monitor gebruiken, maar je kunt ook een willekeurig ander terminal-programma gebruiken, bijv. PuTTY.

De interface begint met het afdrukken van een prompt, zodat de gebruiker weet dat er invoer verwacht wordt, bijv. `ArduinOS 1.0 ready`. Vervolgens kun je invoer lezen, die bestaat uit één of meerdere woorden (*tokens*) gescheiden door één of meerdere spaties of newline-karakters. Het is handig om hiervoor een input-routine te maken.

4.1 Input-routine

Het is belangrijk dat de input-routine *non-blocking* is; dat wil zeggen: je programma moet niet wachten totdat er een hele regel ingelezen kan worden, want dan worden alle lopende processen opgehouden. Deze moeten gewoon doorlopen terwijl de gebruiker aan het typen is. Dit betekent dat je input-routine de input letter voor letter moet inlezen, en in een buffer moet plaatsen. Deze buffer moet `static` gedeclareerd worden, zodat hij behouden blijft tijdens verschillende iteraties van de loop. Je mag er vanuit gaan dat de buffer niet groter hoeft te zijn dan 12 karakters, inclusief terminating zero. N.B.: het wordt afgeraden om de `String` class te gebruiken, aangezien deze erg inefficient is met geheugen.

Om te kijken of de gebruiker een karakter heeft ingetypt, kun je gebruik maken van `Serial.available()`. Staat er geen karakter te wachten, dan ga je gewoon door met het uitvoeren van processen. Dit gaat door totdat het token af is, dus als er een spatie of newline-karakter `'\n'` gelezen wordt. De input-routine moet dan niet de spatie of newline maar een nul-karakter `'\0'` aan de buffer toevoegen. Tevens geeft hij met z'n return-waarde aan dat het token compleet is, zodat het verwerkt kan worden. Daarna kan de buffer gewist worden, om klaar te zijn voor het volgende token.

4.2 Commando's verwerken

Een gelezen token moet vergeleken worden met de beschikbare commando's, en de bijbehorende functie moet aangeroepen worden. Dit kan met een lange rij `if...else if`-instructies, maar het is eleganter om alle beschikbare commando's en de bijbehorende functies in één struct-array onder te brengen, en hier met een for-lus doorheen te lopen, bijvoorbeeld:

```
typedef struct {
  char name[BUFSIZE];
  void (*func)();
} commandType;

static commandType command[] = {
  {"store", &store},
  {"retrieve", &retrieve},
     ⋮
}
```

Hierin is `"store"` de naam van een commando, en `store()` de bijbehorende functie. `&store` is dan het adres van die functie. De lengte van de array vind je met:

```
static int n = sizeof(command) / sizeof(commandType);
```

Om een functie uit de array aan te roepen gebruik je

```
command[i].func();
```

Als een ingetoetst commando niet in de lijst voorkomt, moet er een melding geprint worden en een lijst van beschikbare commando's. Hiervoor gebruik je natuurlijk ook weer de array van commando's. De beschikbare commando's staan in tabel 1. Zoals je ziet hebben sommige commando's nog argumenten, zoals een proces-ID. Deze tokens kun je in de bijbehorende functies op dezelfde manier inlezen. Zorg er wel voor dat ook hier het uitvoeren van processen niet geblokkeerd wordt.

Bouw de command line interface en zorg dat er *stubs* zijn voor de functies voor alle commando's. Zorg dat de argumenten ook ingelezen worden. Laat dit door de docent checken.

5 Checkpoint 3: bestandssysteem

Het bestandssysteem bestaat uit een *file allocation table* (FAT) en een stuk opslagruimte voor de data. Deze worden opgeslagen in het EEPROM (Electronically Erasable Programmable Read-Only Memory). De naam Read-Only Memory is misleidend, want dit geheugen is wel degelijk schrijfbaar, maar de inhoud gaat niet verloren als de voeding van de Arduino wegvalt. Tegelijkertijd is het aantal schrijfoperaties op dit soort geheugen niet onbeperkt; na 100.000 schrijfoperaties bestaat de kans dat het geheugen beschadigd raakt. Wij gebruiken het hier als een simulatie van een magnetische schijf of flash-geheugen.

5.1 File Allocation Table

De FAT kan het handigst als een struct-array geïmplementeerd worden. Per bestand moet bijgehouden worden:

- de bestandsnaam (12 karakters, inclusief terminating zero)
- de beginpositie in het EEPROM (een integer)
- de lengte (een integer)

Tabel 1: Beschikbare commando's in de command line interface

`STORE` *`bestand grootte data`*	sla een bestand op in het bestandssysteem
`RETRIEVE` *`bestand`*	vraag een bestand op uit het bestandssysteem
`ERASE` *`bestand`*	wis een bestand
`FILES`	print een lijst van bestanden
`FREESPACE`	print de beschikbare ruimte in het bestandssysteem
`RUN` *`bestand`*	start een programma
`LIST`	print een lijst van processen
`SUSPEND` *`id`*	pauzeer een proces
`RESUME` *`id`*	hervat een proces
`KILL` *`id`*	stop een proces

Met de instructies `EEPROM.put()` en `EEPROM.get()` kunnen complete structs in één keer van en naar het EEPROM gelezen en geschreven worden. Gebruik het eerste deel van het EEPROM om 10 van deze structs op te slaan. Maak hiervoor functies als `writeFATEntry()` en `readFATEntry()`.
Ook is een variabele (byte) nodig om het aantal aanwezige bestanden bij te houden. Met de class `EERef` maak je een object dat als een gewone variabele (van één byte) gelezen en geschreven kan worden, maar dat zich in het EEPROM bevindt, bijvoorbeeld:

```
EERef noOfFiles = EEPROM[160];
```

Hierna kun je `noOfFiles` gebruiken als een gewone variabele, maar wordt hij opgeslagen op EEPROM-lokatie 160. Tenslotte kun je de rest van het EEPROM gebruiken om de bestandsdata op te slaan. De totale grootte van het EEPROM vind je met `EEPROM.length()`.
In onderstaande drie onderdelen moet telkens in de FAT gezocht worden naar een bestand met een bepaalde naam. Het is handig om hiervoor één functie te schrijven, die het nummer teruggeeft van het bestand in de FAT, of -1 als het niet gevonden wordt.

5.2 Schrijven naar bestand

Met het commando `STORE` moet een bestand naar het bestandssysteem geschreven worden. Het commando heeft als argumenten de naam, de grootte (in bytes) en de data. Maak een functie die een bestand met die naam en grootte alloceert. Check daarvoor eerst of er nog een vrije entry in de FAT is, en zoniet, print een foutmelding. Maak een functie die checkt of een bestand met een bepaalde naam aanwezig is in de FAT. Check daarmee of er al een bestand met de opgegeven naam is, en print in dat geval een foutmelding. Zoniet, zoek dan in de FAT waar op de “schijf” voldoende ruimte is. Dit doe je door de FAT te sorteren op beginpositie en vervolgens telkens te kijken hoeveel bytes verschil er zit tussen het eind van een bestand en het begin van het volgende (of het eind van de “schijf”). Bestanden hoeven immers niet aaneengesloten op de “schijf” te staan. Is er nergens voldoende ruimte, print dan een foutmelding.
Voeg de nieuwe entry toe aan de FAT (`noOfFiles` wordt met 1 opgehoogd). Lees zoveel bytes van de seriële input als de grootte aangeeft, en plaats deze op de juiste plaats in het EEPROM. Om te zorgen dat het systeem blijft wachten op input kun je in de `setup()` de instructie `Serial.setTimeout(-1)` gebruiken. Wis na het lezen van het opgegeven aantal bytes eventuele rest-data in de seriële buffer. Print een melding dat het bestand succesvol is opgeslagen.

5.3 Lezen uit bestand

Het commando `RETRIEVE` moet een bestand uit het bestandssysteem weergeven op de seriële output. Het commando heeft één argument: de bestandsnaam. Check op aanwezigheid van dit bestand met de functie uit §5.2. Is het bestand aanwezig, geef dan de inhoud uit het EEPROM weer op de seriële output. Zoniet, print dan een foutmelding.

5.4 Bestand wissen

Het commando `ERASE` wist een bestand. Het heeft ook weer één argument: de naam van het bestand. Maak een functie die checkt of een bestand van die naam aanwezig is en dit wist uit de FAT (`noOfFiles` wordt met 1 verlaagd), of anders een foutmelding printt.

5.5 Lijst van bestanden weergeven

Het commando `FILES` moet een lijst printen van alle bestanden in het bestandssysteem, met hun grootte in bytes.

5.6 Beschikbare ruimte weergeven

Hert commando `FREESPACE` geeft de maximale grootte van een bestand weer dat nog in het EEPROM geplaatst kan worden. Hiervoor moet je de FAT sorteren op beginpositie, en kijken hoeveel bytes verschil er zit tussen het eind van een bestand en het begin van het volgende (of het eind van de “schijf”). Print de grootste gevonden waarde weer op de seriële output.

Bouw het bestandssysteem en zorg dat alle functionaliteit aanwezig is. Houd er rekening mee dat het lezen en schrijven van bestanden niet alleen mogelijk is vanaf de command line interface, maar later ook vanuit programmacode. Laat je bestandssysteem checken door de docent.

6 Checkpoint 4: geheugenbeheer, variabelen en stack

Het systeem voor het opslaan van variabelen bestaat uit een stuk werkgeheugen en een *memory table*. Er moeten functies komen om variabelen te schrijven, te lezen en te wissen. Deze functies werken met een *stack*: de data die in de variabele wordt opgeslagen, komt van de stack. Andersom wordt bij het teruglezen van een variabele de data op de stack gezet. Ieder proces (zie volgende checkpoint) heeft zijn eigen stack. Omdat we nog geen processen geïmplementeerd hebben, zullen we voor dit checkpoint één stack gebruiken. Later zul je voor ieder proces een aparte stack gebruiken, en zul je de code uit dit checkpoint een klein beetje moeten aanpassen.

6.1 Geheugen

In het werkgeheugen worden variabelen opgeslagen, die van type CHAR (1 byte), INT (2 bytes, big-endian), FLOAT (4 bytes volgens IEEE 754-notatie) of STRING (zero-terminated, variabel aantal bytes). Het geheugen wordt voorgesteld als een global byte array van 256 bytes (zodat het met 1 byte geadresseerd kan worden). Om bij te houden waar in het geheugen de data staat heb je een memory table nodig.

6.2 Memory table

In de memory table wordt per variabele bijgehouden: de naam, het type, het adres, de grootte, en bij welk proces hij hoort. Het is handig om hier weer een struct-array voor te gebruiken van bijv. 25 elementen voor alle processen samen. De naam is 1 byte, en samen met het proces-ID (int) vormt die een unieke combinatie (verschillende processen kunnen dus wel variabelen met dezelfde naam hebben). Houd met een global variabele het aantal in gebruik zijnde variabelen bij, bijv. `noOfVars`.

6.3 Stack

De stack wordt gebruikt als tijdelijke opslag van data, bijvoorbeeld argumenten voor instructies of resultaten van berekeningen. De stack bestaat uit een stuk geheugen (byte array) en een *stack pointer* (SP) die aangeeft waar in het geheugen het volgende element op de stack geplaatst wordt. De basale bewerkingen met de stack zijn het *pushen* van een byte (de SP wordt met 1 opgehoogd) en het *poppen* van een byte (de SP wordt met 1 verlaagd). Hiervoor kun je de volgende code gebruiken:

```
1 #define STACKSIZE 32
2 byte stack[STACKSIZE];
3 byte sp = 0;
4 void pushByte(byte b) {
5   stack[sp++] = b;
6 }
7 byte popByte() {
8   return stack[--sp];
9 }
```

We willen echter geen bytes, maar waardes van de verschillende datatypen (CHAR, INT, FLOAT en STRING) op de stack kunnen opslaan. Daarvoor moet dus niet alleen de waarde, maar ook het type op de stack gezet worden. In §A.1 staat beschreven hoe dat in z'n werk gaat.

6.4 Getallen pushen

Voor de numerieke types `CHAR`, `INT` en `FLOAT` is het handig om functies te maken die een meegegeven getal pushen als losse bytes. Bijvoorbeeld voor de functie `pushInt()` kun je de Arduino-functies `highByte()` en `lowByte()` gebruiken. Voor `pushFloat()` moet je het getal vertalen naar 4 losse bytes. Hiervoor kun je een byte-array en een pointer naar een float naar hetzelfde stukje geheugen laten wijzen:

```
byte b[4];
float *pf = (float *)b;
```
Als je `*pf` dan een waarde geeft, kun je de losse bytes benaderen via `b[]` en (in omgekeerde volgorde) op de stack pushen. Tenslotte moet het type ook gepusht worden.

6.5 Getallen poppen

Voor het poppen van de numerieke types `INT` en `FLOAT` geldt het omgekeerde als voor het pushen: het is handig om functies te hebben die uit de losse bytes weer getallen reconstrueren. Voor `popInt()` kun je gebruik maken van de Arduino-functie `word()`. Voor `popFloat()` kun je dezelfde constructie gebruiken als hierboven beschreven. Een functie `popChar()` is niet nodig; deze is immers identiek aan `popByte()`.
Voor sommige instructies maakt het niet uit of ze als argument een `CHAR`, `INT` of `FLOAT` krijgen, bijvoorbeeld `DIGITALREAD`. Deze instructie heeft als argument het pin-nummer, maar het maakt niet uit welk type dit heeft. Hiervoor is het handig om een functie `popVal()` te maken, die zelf kijkt wat voor type er op de stack staat en kiest of `popByte()`, `popInt()` of `popFloat()` aangeroepen moet worden. Het resultaat wordt altijd als float teruggegeven (dit type kan immers de waarden van de andere bevatten).
Verder is het handig om een *peek*-mogelijkheid te hebben, die het bovenste getal van de stack teruggeeft, maar dit ook op de stack laat staan. Dit zou je met een extra boolean argument kunnen realiseren.

6.6 String pushen

Bij het pushen van een string worden de karakters één voor één op de stack geplaatst, tot en met de terminating zero. Tegelijkertijd moet je tellen wat de lengte van de string was (inclusief terminating zero), en deze lengte moet dan als byte ook gepusht worden. Tenslotte moet het type (`STRING`) gepusht worden.

6.7 String poppen

Bij het poppen van een `STRING`, bijvoorbeeld voor het uitprinten van tekst, moet na het type eerst de lengte gepopt worden. Dan kunnen zoveel bytes gepopt worden als de lengte aangeeft. Het lastige is dat de karakters van de string in omgekeerde volgorde van de stack gepopt worden. Echter, aangezien de bytes ook na het poppen nog in het geheugen staan zolang er geen nieuwe gegevens gepusht zijn, is het mogelijk de SP na het poppen direct als character pointer te gebruiken. Het is handig om hier ook een functie `popString()` voor te schrijven, die de character pointer teruggeeft, voor gebruik met bijvoorbeeld `Serial.print()`.

6.8 Variabele opslaan

Maak een functie die een waarde (getal of string) van de stack leest en deze opslaat in het geheugen. Deze functie heeft de naam (byte) en het proces-ID (int) als argumenten. Check daarvoor eerst of er nog ruimte in de memory table is. Zoniet, print dan een foutmelding. Dan moet je kijken of er al een variabele met die naam en proces-ID bestaat. Zoja, dan moet deze gewist worden uit de memory table (de variabele wordt als het ware “overschreven”). Dit betekent dat alle volgende entries in de memory table een plaatsje omhoog schuiven, en dat `noOfVars` met 1 verlaagd wordt. Vervolgens moet je kijken hoeveel bytes geheugen je nodig hebt, door het type van de stack te poppen. Voor `CHAR`, `INT` en `FLOAT` is het type gelijk aan het benodigde aantal bytes. Voor `STRING` variëert dit, en moet je het volgende element van de stack poppen om de grootte te weten te komen. Dan zoek je naar een plek in het geheugen waar zoveel bytes beschikbaar zijn. Dit gaat op dezelfde manier als het vinden van een plek voor een bestand in het bestandssysteem: sorteer de memory table op adres en kijk telkens naar het verschil tussen het adres plus de grootte van een variabele en het adres van de volgende. Is dit gelijk aan of groter dan het benodigde aantal bytes, dan kan de variabele hier opgeslagen worden. Is er nergens plek, print dan een foutmelding. Vervolgens maak je een nieuwe entry in de memory table met het gevonden adres, pop je het juiste aantal bytes van de stack en schrijf je deze (in omgekeerde volgorde) naar de juiste geheugenplaatsen. Tenslotte wordt `noOfVars` met 1 opgehoogd.

6.9 Variabele teruglezen

Maak een functie die een variabele uit het geheugen ophaalt en op de stack plaatst. Deze functie heeft de naam (byte) en het proces-ID (int) als argumenten. Zoek in de memory table naar de juiste variabele en

proces-ID. Kan dit niet gevonden worden, print dan een foutmelding. De opgeven grootte in de memory table bepaalt het aantal bytes dat je uit het geheugen moet lezen en op de stack moet pushen. In geval van het type `STRING` moet je ook nog de grootte op de stack pushen. Push tenslotte het type op de stack.

6.10 Alle variabelen wissen

Als een proces eindigt, moeten alle variabelen die bij dat proces horen, gewist worden. Maak een functie die dit doet. Deze functie heeft het proces-ID (int) als argument. De functie gaat hele memory table door en wist alle entries die bij dit proces horen. Dit betekent dat alle volgende entries één of meerdere plekjes omhoog schuiven. Ook `noOfVars` wordt aangepast.

Implementeer de functies en schrijf een stukje testcode om te zien of je van alle vier de typen gegevens op de stack kunt pushen, als variabele kunt opslaan, deze weer kunt teruglezen naar de stack en weer van de stack kunt poppen. Laat ook zien dat je een bestaande variabele kunt veranderen. Laat je code door de docent checken.

7 Checkpoint 5: processen

Ieder programma dat door het OS wordt uitgevoerd is een proces. Er kunnen meerdere processen tegelijk uitgevoerd worden (ook meerdere instanties van hetzelfde programma). Processen kunnen gestart, gepauzeerd, hervat en gestopt worden. In de process table wordt de status van elk proces bijgehouden.

7.1 Process table

In de process table wordt van ieder proces bijgehouden: de naam van het programma, een proces-ID, de toestand (`RUNNING` (`'r'`), `PAUSED` (`'p'`) of `TERMINATED` (0)), plus een aantal registers: de program counter (PC), de file pointer (FP), de stack pointer (SP), het adres van het begin van de loop. Tenslotte wordt per proces een aparte stack bijgehouden. Al deze onderdelen kunnen in één struct-array ondergebracht worden van bijv. 10 elementen. Het proces-ID is een uniek nummer dat bij ieder nieuw proces met 1 opgehoogd wordt, dus hier moet ook een teller voor bijgehouden worden (global variabele).

7.2 Proces starten

Het commando `RUN` op de command line start een nieuw proces. Check daarvoor eerst of er ruimte is in de process table. Zoniet, print dan een foutmelding. Het argument van het commando is de naam van een bestand in het bestandssysteem. Gebruik de functie uit §5.2 om te checken of dit bestand bestaat en zoniet, print dan een foutmelding. Als het bestand bestaat kun je de gegevens invullen in de process table: de naam is de opgegeven naam; het proces-ID komt van de teller (die met 1 opgehoogd wordt), de PC is het beginadres van het bestand (uit de FAT), de SP wordt op 0 gezet, en de toestand wordt op `RUNNING` gezet. Print een melding dat het proces succesvol gestart is met het proces-ID.

7.3 Proces pauzeren

Het commando `SUSPEND` pauzeert een lopend proces. Het argument is het proces-ID van het te pauzeren proces. Maak een functie die checkt of dit ID bestaat (en niet `TERMINATED` is), en de index van het proces in de process table teruggeeft. Print anders een foutmelding. Maak ook een functie die een proces in een andere toestand brengt. De argumenten zijn de index in de process table en de gewenste toestand. Print een foutmelding als het proces zich al in die toestand bevindt. Print anders een bevestiging van de nieuwe toestand.

7.4 Proces hervatten

Het commando `RESUME` hervat een gepauzeerd proces. Hiervoor kun je precies dezelfde functies gebruiken als in de vorige paragraaf.

7.5 Proces stoppen

Het commando `KILL` stopt een lopend of gepauzeerd proces. Hiervoor kun je precies dezelfde functies gebruiken als in de vorige paragraaf. Tevens moet je alle variabelen wissen die bij dit proces horen. Hiervoor kun je de functie uit §6.10 gebruiken.

7.6 Lijst van processen weergeven

Het commando `LIST` print een lijst van lopende en gepauzeerde processen, met hun ID, status en naam. Doorloop hiervoor de hele process table en print de gegevens van ieder bestand dat niet `TERMINATED` is.

Implementeer de process table en de commando's. Pas de stack-functies aan zodat ze werken met een aparte stack voor ieder proces. Laat aan de docent zien dat je processen kunt starten, pauzeren en stoppen door telkens de lijst van processen te tonen.

8 Checkpoint 6: instructies uitvoeren

Nu we alle "infrastructuur" op orde hebben, komen we toe aan de belangrijkste taak van ArduinOS: het uitvoeren van instructies in de bytecode-taal. Deze staat beschreven in appendix A. Om te testen is ook een aantal voorbeeldprogramma's beschikbaar in bytecode-taal. Deze kunnen in het bestandssysteem geplaatst worden met behulp van het programma `convert` dat op de host-computer draait, zie appendix B.

8.1 Processen laten lopen

Van ieder lopend proces moet telkens één instructie worden uitgevoerd. Maak een functie (bijv. `runProcesses()`) die de process table doorloopt en voor ieder proces met de toestand `RUNNING` een instructie inleest en uitvoert (zie hieronder). Zorg dat deze functie iedere iteratie van de `loop()` uitgevoerd wordt. Zorg ook dat deze functie herhaaldelijk wordt uitgevoerd als de command line interface staat te wachten op argumenten van een commando, bijvoorbeeld de bestandsnaam nadat het commando `store` gegeven is.

8.2 Instructie inlezen

Maak voor het inlezen en uitvoeren van instructies een functie (bijv. `execute()`) die als argument de index van het proces in de process table meekrijgt, zodat de juiste PC en stack gebruikt kunnen worden (ieder proces heeft immers zijn eigen PC en stack). De instructies en hun argumenten staan in een bestand in het bestandssysteem, in het EEPROM. De PC wijst altijd naar de eerstvolgende instructie. Die moet ingelezen worden en geïnterpreteerd. Dit gaat het handigst met een `switch...case`-structuur. Voor iedere instructie moet een stukje Arduino-code geschreven worden dat eventuele argumenten inleest, en de instructie uitvoert. Ook moet de PC telkens aangepast worden.
De complete instructieset is weergegeven in appendix C. De instructies staan als numerieke constanten in de header-file `instruction_set.h`. Hier geven we enkele tips voor de implementatie van de instructies.

8.3 Waarden op de stack plaatsen

De instructies `CHAR`, `BYTE` en `FLOAT` plaatsen hun argument op de stack. Voor deze drie instructies samen kun je één functie schrijven, die zoveel bytes als het type aangeeft uit het EEPROM leest en op de stack pusht. Voor de instructie `STRING` moet je zoveel bytes lezen en pushen totdat je een nul-karakter (`'\0'`) tegenkomt. De 0 moet ook gepusht worden, gevolgd door de lengte van de string (inclusief terminating zero) en het type.

8.4 Console output

`PRINT` en `PRINTLN` poppen één waarde van een willekeurig type en printen deze uit. `PRINTLN` voegt aan het eind nog een newline-karakter (`'\n'`) toe. De wijze van uitprinten verschilt per type: een `CHAR` wordt als een karakter geprint, dus niet als een getal. `INT` en `FLOAT` worden als getallen geprint, en `STRING` als tekst.

8.5 Variabelen

`GET` en `SET` verplaatsten data respectievelijk van de stack naar een variabele en terug. De naam van de variabele (1 byte) volgt op de instructie. Gebruik de functies uit §6.8 en §6.9 om deze instructies te implementeren.

8.6 Unaire operatoren

Unaire operatoren zijn instructies die 1 waarde van de stack poppen er er ook weer 1 pushen. Wat ze met de waarde doen verschilt per instructie. Voor dit soort operatoren kun je één functie schrijven die een waarde popt, een bewerking doet en de uitkomst weer pusht. Gebruik bijv. een `switch...case`-constructie om te kiezen welke bewerking er moet gebeuren. Het type van de teruggeplaatste waarde verschilt per bewerking: soms is het gelijk aan het type van de gebruikte waarde, soms is het een vastgesteld type. Dit is beschreven in tabel 2. Zorg dat je bij het pushen het juiste type gebruikt.

8.7 Binaire operatoren

Binaire operatoren zijn instructies die 2 waarden van de stack poppen en er weer 1 pushen. Ook hier kun je één functie maken die alle binaire operatoren afhandelt. Let op: de tweede waarde staat boven de eerste op de stack, dus als je $x - y$ moet uitrekenen, is de eerste waarde die van de stack popt y en de tweede x. Het type van de teruggeplaatste waarde is vaak het meest precieze van de twee gebruikte typen. Dus als je bijv. een `CHAR` en een `FLOAT` gebruikt, zal de uitkomst van type `FLOAT` zijn. Met een `INT` en een `CHAR` is het resultaat van type `INT`. De binaire operatoren staan in tabel 3. Zorg dat je bij het pushen het juiste type gebruikt.

Zorg dat je OS instructies kan uitvoeren en implementeer zoveel mogelijk van de instructies in appendix C, maar in ieder geval de instructies `CHAR`, `INT`, `FLOAT`, `STRING`, `SET`, `GET`, `PRINTLN`, `PLUS`, `MINUS`, `INCREMENT`, `DECREMENT`. Laat dit checken door de docent.

9 Checkpoint 7: flow control, timing, file I/O en forking

Om je programma's interessante dingen te kunnen laten doen, moet het instructies kunnen herhalen en beslissingen kunnen nemen; met andere woorden: flow control. Hiervoor heb je instructies nodig als `STOP`, `IF`, `LOOP` en `WHILE`. Daarnaast kan het nuttig zijn om pauzes te kunnen inlassen (timing) en gegevens van of naar een bestand te kunnen lezen of schrijven (file I/O). Tot slot zou je vanuit één proces een ander proces kunnen opstarten (forking).

Tabel 2: Unaire operatoren

operator	popt type	pusht type
INCREMENT	CHAR, INT, FLOAT	zelfde type
DECREMENT	CHAR, INT, FLOAT	zelfde type
UNARYMINUS	CHAR, INT, FLOAT	zelfde type
ABS	CHAR, INT, FLOAT	zelfde type
SQ	CHAR, INT, FLOAT	zelfde type
SQRT	CHAR, INT, FLOAT	zelfde type
ANALOGREAD	CHAR, INT, FLOAT	INT
DIGITALREAD	CHAR, INT, FLOAT	CHAR
LOGICALNOT	CHAR, INT, FLOAT	CHAR
BITWISENOT	CHAR, INT	zelfde type
TOCHAR	INT, FLOAT	CHAR
TOINT	CHAR, FLOAT	INT
TOFLOAT	CHAR, INT	FLOAT
ROUND	CHAR, INT, FLOAT	INT
FLOOR	CHAR, INT, FLOAT	INT
CEIL	CHAR, INT, FLOAT	INT

Tabel 3: Binaire operatoren

operator	popt type (2×)	pusht type
PLUS	CHAR, INT, FLOAT	max. van beide
MINUS	CHAR, INT, FLOAT	max. van beide
TIMES	CHAR, INT, FLOAT	max. van beide
DIVIDEDBY	CHAR, INT, FLOAT	max. van beide
MODULUS	CHAR, INT	max. van beide
EQUALS	CHAR, INT, FLOAT	CHAR
NOTEQUALS	CHAR, INT, FLOAT	CHAR
LESSTHAN	CHAR, INT, FLOAT	CHAR
LESSTHANOREQUALS	CHAR, INT, FLOAT	CHAR
GREATERTHAN	CHAR, INT, FLOAT	CHAR
GREATERTHANOREQUALS	CHAR, INT, FLOAT	CHAR
MIN	CHAR, INT, FLOAT	max. van beide
MAX	CHAR, INT, FLOAT	max. van beide
POW	CHAR, INT, FLOAT	max. van beide
LOGICALAND	CHAR, INT, FLOAT	CHAR
LOGICALOR	CHAR, INT, FLOAT	CHAR
LOGICALXOR	CHAR, INT, FLOAT	CHAR
BITWISEAND	CHAR, INT	max. van beide
BITWISEOR	CHAR, INT	max. van beide
BITWISEXOR	CHAR, INT	max. van beide

9.1 STOP

De STOP-instructie moet alle variabelen van het huidige proces wissen (zie §6.10) en de status op TERMINATED zetten.

9.2 IF, ELSE en ENDIF

De IF-instructie heeft één argument, een byte die volgt op de instructie zelf. Het is eigenlijk een jump-if-zero; dat wil zeggen: er moet het meegegeven aantal bytes vooruit gesprongen worden als er een waarde gelijk aan nul op de stack staat. Deze waarde (van type CHAR, INT of FLOAT) moet op de stack blijven staan (of na het poppen weer gepusht), omdat er mogelijk nog een ELSE-instructie volgt die de waarde ook nodig heeft.
De ELSE-instructie heeft ook één argument en is analoog aan de IF-instructie. Het is eigenlijk een jump-if-not-zero; er moet het meegegeven aantal bytes vooruit gesprongen worden als er een waarde *niet* gelijk aan nul op de stack staat. Deze waarde moet op de stack blijven staan (of na het poppen weer gepusht), omdat er nog een ENDIF-instructie volgt.
De ENDIF doet niets behalve de waarde die op de stack is blijven staan, eraf halen.

9.3 LOOP en ENDLOOP

De LOOP-instructie zet de huidige waarde van de PC in het loop-register van dit proces (zie §7.1). Dit is het punt waar het programma verder moet gaan als de instructie ENDLOOP gegeven wordt. Die instructie maakt de PC gelijk aan de waarde van het loop-register.

9.4 WHILE en ENDWHILE

De WHILE-instructie heeft twee argumenten, die als bytes volgen op de instructie zelf. Dit zijn het aantal bytes in de conditie-code, die voorafgaat aan de WHILE-instructie, en het aantal bytes tussen WHILE en de bijbehorende ENDWHILE (de body). Het resultaat dat door de conditie-code op de stack geplaatst wordt, bepaalt of de body uitgevoerd moet worden. Daarna moet de conditie-code opnieuw uitgevoerd worden om een nieuwe test mogelijk te maken. De WHILE-instructie popt één waarde (van type CHAR, INT of FLOAT) en als deze gelijk aan 0 is, wordt de PC opgehoogd met het aantal bytes in de body (het tweede argument) plus 1. Zoniet, dan wordt een byte op de stack gepusht die gelijk is aan de twee argumenten samen plus 4. Dit is het aantal bytes dat

ENDWHILE moet terugspringen om weer bij het begin van de conditiecode te komen. ENDWHILE moet dus één byte van de stack poppen en de PC met zoveel verlagen.

9.5 Timing

Het gebruik van de instructie DELAY wordt afgeraden, aangezien hiermee het OS en alle andere processen stilgelegd worden. Voor compleetheid is hij toch in de instructieset opgenomen. De instructie popt een waarde (van type CHAR, INT of FLOAT) en wacht het daardoor aangegeven aantal milliseconden.
Beter is om de instructie MILLIS te gebruiken in combinatie met DELAYUNTIL. MILLIS pusht de huidige waarde van de milliseconden-counter als INT op de stack. Daar kan dan een getal bij opgeteld worden. DELAYUNTIL popt een waarde (van type CHAR, INT of FLOAT) en als deze waarde groter is dan de huidige waarde van de milliseconden-counter, wordt de waarde teruggepusht en de PC met 1 verlaagd.

9.6 File I/O

De OPEN-instructie popt 2 waarden van de stack: een STRING (de bestandsnaam) en een waarde van type CHAR, INT of FLOAT (het aantal bytes). Hiermee wordt een bestaand of een nieuw bestand geopend (gebruik de functie uit §5.2). Let op dat het nu wel mogelijk moet zijn om naar een bestaand bestand te schrijven. Als het bestand al bestaat wordt het opgegeven aantal bytes genegeerd. De FP wordt gelijk gemaakt aan het adres van het begin van het bestand in het EEPROM.
De WRITE-instructie schrijft een waarde van de stack naar het bestand. Hierbij worden alleen de bytes van de waarde geschreven (in omgekeerde volgorde), niet het type. De FP wordt opgehoogd met het aantal geschreven bytes.
De instructies READCHAR, READINT en READFLOAT lezen respectievelijk 1, 2 of 4 bytes uit het EEPROM en pushen deze op de stack, gevolgd door het type. De FP wordt opgehoogd met het aantal gelezen bytes. Je kunt hiervoor één functie gebruiken die als extra argument het te lezen type heeft, bijvoorbeeld de functie die je in §8.3 gemaakt hebt voor het inlezen van numerieke waarden.
De READSTR-instructies leest bytes uit het EEPROM totdat je een nul-karakter ('\0') tegenkomt. De bytes worden, inclusief nul-karakter, op de stack gepusht, gevolgd door de lengte (inclusief het nul-karakter) en het type (STRING). De FP wordt opgehoogd met het aantal gelezen bytes. Je kunt hiervoor dezelfde functie gebruiken die je in §8.3 gemaakt hebt voor het inlezen van een STRING.
De CLOSE instructie doet helemaal niets, en is alleen aanwezig voor compleetheid. Als in de toekomst een ander medium dan het EEPROM gebruikt zou worden voor data-opslag, zou deze instructie kunnen zorgen dat een te schrijven bestand netjes afgesloten wordt.

9.7 Forking

De FORK-instructie popt een STRING en start het bestand met die naam als een nieuw proces. Hiervoor kun je de functie uit §7.2 gebruiken. Geef een foutmelding als er geen ruimte in de process table meer is of het bestand niet gevonden kan worden. Het proces-ID van het nieuwe proces wordt als INT gepusht.
De WAITUNTILDONE-instructie popt een waarde (van type CHAR, INT of FLOAT) en kijkt of het proces met dat proces-ID TERMINATED is. Zoniet, dan wordt de waarde weer gepusht en de PC met 1 verlaagd.

Implementeer zoveel mogelijk van de beschreven instructies, maar in ieder geval STOP, IF, ELSE, ENDIF, LOOP, ENDLOOP, WHILE, ENDWHILE, MILLIS en DELAYUNTIL. Gebruik de demonstratieprogramma's om te laten zien dat je OS meerdere programma's tegelijk kan uitvoeren. Laat dit checken door de docent.

A Bytecode

De bytecode-taal bestaat uit een aantal instructies, elk van 1 byte (zie tabel in appendix C). Een instructie kan ook één of meer argumenten hebben, bijvoorbeeld de naam van een variabele (ook 1 byte), die volgen op het instructiebyte. De instructies en hun argumenten worden byte voor byte gelezen uit een bestand in het bestandssysteem en uitgevoerd. Voor iedere gelezen byte wordt de *program counter* (PC) met 1 opgehoogd, en verwijst dus altijd naar de volgende instructie.

A.1 Stack

Voor tussentijdse opslag van data wordt gebruik gemaakt van een *stack*. Hierop kunnen bytes *gepusht* worden, en daarna in omgekeerde volgorde weer *gepopt*. Bijvoorbeeld, om een waarde aan een variabele toe te kennen, gebruik je in C:

```
int a = 300;
```

In bytecode wordt dit:

`INT 1 44` plaats het getal $1 \times 256 + 44 = 300$ als 2-byte integer op de stack.
`SET 'a'` haal een getal van de stack af en sla dit op in het geheugen als variabele `a`.

Na de instructie `INT 1 44` ziet de stack er als volgt uit:

INT ← bovenkant van de stack
44
1

Zoals je ziet wordt eerst de hoge byte, dan de lage byte, en dan het type op de stack gepusht. Het type staat bovenaan, zodat bij het er weer afhalen eerst het type gepopt wordt en het duidelijk is uit hoeveel bytes het getal zelf bestaat. Na de instructie `set 'a'` is de stack weer leeg.

Operatoren zoals $+$, $-$, $\times$ of $/$ zijn ook instructies. Deze halen 2 getallen van de stack, voeren daarmee een bewerking uit, en plaatsen het resultaat weer terug. Bijvoorbeeld, de C-code:

```
int b = a + 3;
```

wordt in bytecode:

`GET 'a'` haal de waarde van de variabele a uit het geheugen en plaats deze op de stack.
`INT 0 3` plaats een integer met waarde $0 \times 256 + 3 = 3$ op de stack.
`PLUS` haal 2 getallen van de stack, tel ze bij elkaar op, en plaats het resultaat weer terug.
`SET 'b'` haal een getal van de stack af en sla dit op in het geheugen als variabele `b`.

Na de tweede instructie ziet de stack er als volgt uit:

INT ← bovenkant van de stack
3
0
INT
44
1

De twee getallen worden vervolgens door de `PLUS`-operator van de stack gepopt en de som weer gepusht.

Op de stack kunnen ook strings worden opgeslagen. Omdat deze van variabele lengte zijn, moet ook de lengte op de stack gepusht worden. Bij het poppen krijg je dan eerst het type, dan de lengte, en dan weet je hoeveel bytes je van de stack moet halen om de hele string te krijgen. Bijvoorbeeld, na het uitvoeren van de volgende bytecode:

`STRING 'H' 'e' 'l' 'l' 'o' 0` string met terminating zero

staat er het volgende op de stack:

STRING ← bovenkant van de stack
6 ← lengte van de string inclusief terminating zero
0 ← terminating zero
'o'
'l'
'l'
'e'
'H'

Zoals je ziet staat de string in omgekeerde volgorde op de stack. Bij het gebruiken van een string van de stack (bijvoorbeeld met de instructie `PRINT`) moet je hiermee rekening houden.

A.2 Console output

Vaak moet er data van de stack naar de console geschreven worden. Hiervoor zijn de instructies `PRINT` en `PRINTLN`. Deze instructies halen een data-item van de stack en drukken dat af op de console (`PRINTLN` drukt daarna nog een newline-karakter `\n` af). Hoe dit gebeurt, hangt af van het datatype (`CHAR`, `INT`, `FLOAT` of `STRING`). Dit type staat bovenaan de stack, en moet eerst gepopt worden.

Voor de numerieke typen (`CHAR`, `INT` of `FLOAT`) is het type gelijk aan het aantal bytes (1, 2 of 4) die samen het getal vormen. Dit aantal bytes moet gepopt worden, en het getal moet daaruit gereconstrueerd worden. Een `CHAR` wordt geprint als het bijbehorende ASCII-karakter. `INT` en `FLOAT` worden als decimale getallen geprint, bijvoorbeeld:

`CHAR 'a' PRINTLN`	print de letter `a`.
`INT 1 7 PRINTLN`	print het getal $1 \times 256 + 7 = 263$.
`FLOAT 66 246 230 102 PRINTLN`	print het getal 123.45.
`STRING 't' 'e' 's' 't' 0 PRINTLN`	print de string `"test"`.

Voor het type `STRING` staat de lengte onder het type op de stack, en dit geeft aan hoeveel bytes vervolgens gepopt moeten worden. Bij het poppen komen de tekens van de string in omgekeerde volgorde van de stack. Deze moeten dus eerst in een buffer geplaatst worden voordat ze afgedrukt kunnen worden.[1]
N.B. Doordat de lengte als één byte op de stack staat, is de lengte van een string maximaal 255 tekens, inclusief de terminating zero.

A.3 Flow control

A.3.1 Stop

De instructie `STOP` beëindigt het proces. Het geheugen dat ingenomen wordt door alle variabelen die bij dit proces horen, moet vrijgegeven worden en de status van het proces moet op `TERMINATED` gezet worden. Als het programma niet uit een oneindige lus bestaat, moet deze instructie als laatste in het programma zijn opgenomen. De instructie kan ook eerder in het programma voorkomen, bijvoorbeeld in een if-constructie.

A.3.2 If

De instructie `IF` heeft één argument: `lengthOfTrueCode`, het aantal bytes in de *body* van de if-stuctuur. De instructie kijkt naar de waarde op de stack, maar laat deze op de stack staan (*peek*). Als deze waarde gelijk aan 0 is, wordt de PC opgehoogd met de waarde van `lengthOfTrueCode` en springt het proces dus zoveel bytes verder. Zoniet, dan gebeurt er niets en wordt gewoon de volgende instructie uitgevoerd.
Later in het programma kan de instructie `ELSE` voorkomen. Deze heeft ook één argument: `lengthOfFalseCode`, het aantal bytes in de body van de else-structuur. Ook deze instructie kijkt naar de waarde op de stack (de waarde blijft staan) en maar springt alleen zoveel bytes verder als die waarde *niet* gelijk aan 0 is. Is de waarde wel 0, dan gebeurt er niets en wordt gewoon de volgende instructie uitgevoerd.
De if-constructie (met of zonder else) wordt altijd afgesloten met de instructie `ENDIF`. Deze popt één waarde van de stack maar doet hier niets mee. Bijvoorbeeld:

`INT 0 3 SET 'a'`	sla getal 3 op als variabele `a`.
`GET 'a' INT 0 5 EQUALS`	vergelijk `a` met 5 en zet het resultaat (0, false) op de stack.
`IF 7`	als het resultaat gelijk aan 0 was, spring 7 bytes verder.
`STRING 'T' 'r' 'u' 'e' 0 PRINTLN`	druk “True” af (wordt niet uitgevoerd).
`ELSE 8`	als het resultaat niet gelijk aan nul was, spring 8 bytes verder.
`STRING 'F' 'a' 'l' 's' 'e' 0 PRINTLN`	druk “False” af.
`ENDIF`	haal het resultaat van de stack.
`STOP`	beëindig het programma.

N.B.: Doordat de argumenten `lengthOfTrueCode` en `lengthOfFalseCode` beide één byte zijn, bevatten de body's van `IF` en `ELSE` maximaal 256 bytes.

A.3.3 Loop

De instructie `LOOP` begint een stuk code dat alsmaar herhaald wordt (vergelijkbaar met de functie `loop()` in Arduino C). De instructie zet de huidige waarde van de PC in een loop-register dat per proces wordt bijgehouden. De instructie `ENDLOOP` maakt de PC gelijk aan de waarde in het loop-register. Het proces gaat dan verder met de instructie die volgde op de instructie `LOOP`. Bijvoorbeeld:

`INT 0 0 SET 'i'`	sla getal 0 op als variabele `i`.
`LOOP`	sla de waarde van de PC op in het loop-register.

[1] Omdat bij het poppen van data van de stack, de data in het geheugen blijven staan totdat ze overschreven worden, is het ook mogelijk (en simpeler) om na het poppen de stack pointer te typecasten naar een character pointer (`char *`) en de string daarmee direct te printen.

`GET 'i' INCREMENT SET 'i'`	verhoog variabele `i` met 1.
`GET 'i' PRINTLN`	druk `i` af.
`ENDLOOP`	zet de PC op de waarde van het loop-register; het proces gaat verder na `LOOP`.

A.3.4 While

De while-constructie blijft een stuk code (de *body*) herhalen zolang het resultaat op de stack van een ander stuk code (de *conditie-code*) niet gelijk aan 0 is. In het programma komt eerst de conditie-code, gevolgd door de instructie `WHILE`. Deze heeft 2 argumenten: `lengthOfConditionCode`, het aantal bytes in de conditie-code, die voorafgaat aan de `WHILE`-instructie; en `lengthOfRepeatedCode`, het aantal bytes in de body. De instructie popt 1 waarde van de stack. Als deze gelijk aan 0 is, dan wordt de PC met `lengthOfRepeatedCode` + 1 opgehoogd. Hierdoor wordt de body overgeslagen. Is de waarde niet gelijk aan 0, dan wordt een waarde van `lengthOfConditionCode` + `lengthOfConditionCode` + 4 als één byte op de stack gepusht. Het programma gaat verder met de volgende instructie.
De constructie wordt afgesloten met de instructie `ENDWHILE`. Deze popt één byte van de stack en trekt deze van de PC af. Hierdoor wordt naar het begin van de conditie-code gesprongen, zodat deze opnieuw uitgevoerd wordt. Bijvoorbeeld:

`INT 0 0 SET 'i'`	sla getal 0 op als variabele `i`. Hierna begint de conditie-code.
`GET 'i' INT 0 5 LESSTHAN`	kijk of `i` kleiner is dan 5 en zet het resultaat op de stack.
`WHILE 6 8`	als het resultaat gelijk aan 0 is, spring dan 8 + 1 = 9 bytes verder (naar `STOP`); zoniet, zet dan 6 + 8 + 4 = 18 als byte op de stack.
`GET 'i' INCREMENT SET 'i'`	verhoog variabele `i` met 1.
`GET 'i' PRINTLN`	druk `i` af.
`ENDWHILE`	haal een byte van de stack en spring zoveel (in dit geval 17) plaatsen terug.
`STOP`	beëindig het programma.

Deze while-loop wordt 5× uitgevoerd, waarna het programma stopt.
N.B.: Doordat de het aantal terug te springen plaatsen als één byte op de stack geplaatst wordt, zijn de conditiecode en de body samen maximaal 251 bytes.

A.4 Delays

Om een pauze in te lassen, kan de instructie `DELAY` gebruikt worden. Deze instructie popt 1 getal van de stack van type `INT`: het aantal milliseconden dat gewacht moet worden. Echter, omdat daarmee het hele OS en alle andere draaiende processen stilgelegd worden, wordt het gebruik van deze instructie afgeraden. Beter is om `DELAYUNTIL` in combinatie met `MILLIS` te gebruiken. `MILLIS` zet de huidige waarde van de milliseconden-counter als `INT` op de stack. Hier kan men met `PLUS` een aantal milliseconden bij optellen. `DELAYUNTIL` popt 1 getal van de stack van type `INT`. Als de milliseconden-counter deze waarde nog niet bereikt heeft, wordt het getal weer teruggepusht en wordt de PC met 1 verlaagd. Hierdoor blijft het proces op deze instructie 'hangen' totdat de gewenste tijd verstreken is, zonder andere processen op te houden. Bijvoorbeeld:

`CHAR LED_BUILTIN SET 'p'`	maak de variabele `p` gelijk aan symbolische constante `LED_BUILTIN`.
`CHAR 1 SET 'o'`	maak de variabele `o` gelijk aan 1.
`GET 'p' CHAR OUTPUT PINMODE`	Arduino C: `pinMode(p, OUTPUT)`
`LOOP`	begint eindeloze loop
`GET 'p' GET 'o' DIGITALWRITE`	Arduino C: `digitalWrite(p, o)`
`GET 'o' LOGICALNOT SET 'o'`	inverteer variabele `o` (1 → 0, 0 → 1).
`MILLIS INT 1 244 PLUS`	tel 1 × 256 + 244 = 500 op bij de huidige milliseconden-counter.
`DELAYUNTIL`	wacht tot de counter deze waarde bereikt heeft (duurt 0,5 s).
`ENDLOOP`	spring terug naar het begin van de loop.

Dit programma laat de ingebouwde LED van de Arduino knipperen met een periode van 1 s.
N.B.: Omdat de waarde van de milliseconden-counter als 16-bits integer wordt opgeslagen, reset deze iedere 65 s. Voor echte toepassingen is dit niet handig; voor demonstratiedoeleinden is dit geen probleem.

A.5 File I/O

Vanuit een programma kan er gelezen worden van en geschreven worden naar bestanden in het bestandssysteem. Er kan per proces maar één file tegelijk worden gebruikt, en de *file pointer* (FP) wordt bijgehouden in een register. Met de instructie `OPEN` wordt een bestand geopend. Deze instructie leest 2 data-items van

de stack: de bestandsnaam (zero-terminated `STRING`) en de lengte van het bestand. De lengte wordt alleen gebruikt als er nog geen bestand met de aangegeven naam bestaat; dan wordt er een nieuw bestand van die lengte gemaakt in het bestandssysteem. Bestaat er wel al een bestand met die naam, dan wordt dat gebruikt. De FP wordt gezet naar het begin van het (nieuwe of bestaande) bestand.
Met `WRITE` kan data naar een bestand geschreven worden, op dezelfde manier als `PRINT` (zie §A.2). Deze instructie popt 1 data-item van de stack, dat van een willekeurig type mag zijn (`CHAR`, `INT`, `FLOAT` of `STRING`), en schrijft dat naar het bestand. De gebruiker is er zelf verantwoordelijk voor om niet meer data te schrijven dan in het bestand past.
Voor het lezen zijn er de instructies `READCHAR`, `READINT`, `READFLOAT` en `READSTRING`. De eerste drie lezen respectievelijk 1, 2 of 4 bytes uit het bestand en plaatsen deze met het aangeven datatype op de stack. De laatste leest een `STRING` uit het bestand tot en met de terminating zero en plaatst deze op de stack.
Voor de volledigheid is er nog de instructie `CLOSE`, die helemaal niets doet. Het is niet nodig bestanden af te sluiten; de data worden meteen naar het EEPROM geschreven. Bijvoorbeeld:

`STRING 'n' 'e' 'w' '_' 'f' 'i' 'l' 'e' 0 SET 's'`	zet de string `"new_file"` in variabele s.
`GET 's'`	zet de inhoud van de variabele op de stack.
`INT 0 20`	zet het getal $0 \times 256 + 20 = 20$ als integer op de stack.
`OPEN`	open een nieuw bestand `new_file` van 20 bytes.
`STRING 't' 'e' 'x' 't' '\n' 0 WRITE`	schrijf de string `"text\n"` naar het bestand
`CHAR 'a' WRITE`	schrijf het karakter `a` naar het bestand
`INT 98 99 WRITE`	schrijf de integer $98 \times 256 + 99 = 25187$ naar het bestand.
`FLOAT 61 204 204 205 WRITE`	schrijf de float 0.1 naar het bestand.
`GET 's'`	zet de bestandsnaam weer op de stack.
`INT 0 20`	zet het getal $0 \times 256 + 20 = 20$ als integer op de stack.
`OPEN`	open het bestand opnieuw; het getal 20 wordt nu genegeerd.
`READSTRING PRINTLN`	lees een zero-terminated string uit het bestand en druk deze af.
`READCHAR PRINTLN`	lees een byte uit het bestand en druk deze als character af.
`READINT PRINTLN`	lees 2 bytes uit het bestand en druk deze als integer af.
`READFLOAT PRINTLN`	lees 4 bytes uit het bestand en druk deze als float af.
`STOP`	beëindig het programma.

A.6 Forking

Het is mogelijk vanuit een programma een ander programma (of hetzelfde) op te starten. Beide programma's draaien dan naast elkaar. Hiervoor is de instructie `FORK`. Deze popt een `STRING` van de stack en start dit bestand als nieuw proces. Het proces-ID wordt als `INT` op de stack geplaatst. Vervolgens wordt gewoon de volgende instructie uitgevoerd, terwijl het andere proces ook loopt.
Als het nodig is om te wachten tot het andere proces klaar is, kan de instructie `WAITUNTILDONE` gebruikt worden. Deze popt een `INT` van de stack en kijkt of het proces met dat ID de status `TERMINATED` heeft. Zoja, dan gebeurt er niets en wordt gewoon de volgende instructie uitgevoerd. Zonee, dan wordt de waarde weer op de stack gepusht en wordt de PC met 1 verlaagd, zodat het proces op deze instructie blijft 'hangen' totdat het andere proces is afgelopen. Bijvoorbeeld:

`STRING 'h' 'e' 'l' 'l' 'o' ',' 0`	zet de string `"hello,"` op de stack.
`PRINTLN`	print de string die op de stack staat.
`STRING 'w' 'o' 'r' 'l' 'd' 0 SET 's'`	zet de string `"world"` in de variabele `s`.
`GET 's'`	zet de inhoud van de variabele weer op de stack.
`INT 0 10`	zet het getal 10 als integer op de stack.
`OPEN`	maak een nieuw bestand `world` van 10 bytes.
`GET 's' WRITE`	schrijf de tekst `world` naar het bestand.
`CHAR PRINTLN WRITE`	schrijf de byte-waarde van de instructie `PRINTLN` naar het bestand.
`CHAR STOP WRITE`	schrijf de byte-waarde van de instructie `STOP` naar het bestand.
`GET 's' FORK`	start het programma met de naam `world`.
`WAITUNTILDONE`	wacht tot het andere programma klaar is.
`STRING 'b' 'y' 'e' 0 PRINTLN`	print de string `"bye"`.
`STOP`	beëindig het programma.

Dit programma print de string `hello,` en maakt vervolgens een nieuw bestand met instructies om de string `world` te printen. Dit bestand wordt vervolgens als programma uitgevoerd. Het eerste programma wacht tot het tweede klaar is, en print tenslotte de string `bye`. De output zal dus zijn:

```
hello
world
bye
```

B Byte code converter

Om programma's in bytecode-taal in het bestandssysteem te plaatsen is het programma `convert` beschikbaar. Start dit programma als volgt:

```
convert <file> <serial port>
```

Hier is `<file>` de naam van het tekstbestand met het gewenste programma en `<serial port>` de naam van de seriële poort waar de Arduino op aangesloten is, bijv. `COM3` op een PC of `/dev/tty.usbmodem141101` op een Mac.
Dit programma vertaalt de instructies in een tekstbestand naar een binair bestand (één byte per instructie), geeft het `store`-commando aan de Arduino en verstuurt de data, zodat het in het bestandssysteem opgeslagen wordt. Merk op dat dit programma ook de mogelijkheid geeft om data van type `CHAR`, `INT`, `FLOAT` en `STRING` makkelijker te noteren. Dit hoeft niet als losse bytes, maar kan als complete getallen:

tekst	wordt geconverteerd naar
'a'	CHAR 0x61
123	INT 0x00 0x7b
1.234	FLOAT 0x3f 0x9d 0xf3 0xb6
"hallo\n"	STRING 0x68 0x61 0x6c 0x6c 0x6f 0x0a 0x00

C Instructieset

In onderstaande tabel wordt de functie van iedere instructie aangeven, en welke argumenten de instructie heeft, wat er gepopt wordt en wat er gepusht wordt. "waarde" geeft aan: een waarde van type `CHAR`, `INT`, `FLOAT` of `STRING`. "getal" geeft aan: een waarde van type `CHAR`, `INT` of `FLOAT`.

instructie	argumenten	popt type	pusht type	betekenis
CHAR	1 byte		CHAR	push 1 byte gevolgd door CHAR
INT	2 bytes		INT	push 2 bytes gevolgd door INT
STRING	zero-terminated string		STRING	push alle bytes incl. terminating zero, gevolgd door lengte (1 byte) en STRING
FLOAT	4 bytes		FLOAT	push 4 bytes gevolgd door FLOAT
SET	1 byte	waarde		sla waarde op als variabele
GET	1 byte		waarde	push variabele op stack
INCREMENT		getal	getal	vermindert met 1
DECREMENT		getal	getal	verhoogt met 1
PLUS		2×getal	getal	$x + y$
MINUS		2×getal	getal	$x - y$
TIMES		2×getal	getal	$x \times y$
DIVIDEDBY		2×getal	getal	x/y
MODULUS		2×getal	getal	x mod y
UNARYMINUS		getal	getal	$-x$
EQUALS		2×getal	CHAR	1 als $x = y$, anders 0
NOTEQUALS		2×getal	CHAR	1 als $x \neq y$, anders 0
LESSTHAN		2×getal	CHAR	1 als $x < y$, anders 0
LESSTHANOREQUALS		2×getal	CHAR	1 als $x \leq y$, anders 0
GREATERTHAN		2×getal	CHAR	1 als $x > y$, anders 0
GREATERTHANOREQUALS		2×getal	CHAR	1 als $x \geq y$, anders 0

instructie	argumenten	popt type	pusht type	betekenis
LOGICALAND		2×getal	CHAR	1 als $x \wedge y$, anders 0
LOGICALOR		2×getal	CHAR	1 als $x \vee y$, anders 0
LOGICALXOR		2×getal	CHAR	1 als $x \oplus y$, anders 0
LOGICALNOT		getal	CHAR	1 als $x = 0$, anders 0
BITWISEAND		2×CHAR of INT	CHAR of INT	`x & y`
BITWISEOR		2×CHAR of INT	CHAR of INT	`x | y`
BITWISEXOR		2×CHAR of INT	CHAR of INT	`x ^ y`
BITWISENOT		2×CHAR of INT	CHAR of INT	`~x`
TOCHAR		INT of FLOAT	CHAR	type-conversie
TOINT		CHAR of FLOAT	INT	type-conversie
TOFLOAT		INT of CHAR	FLOAT	type-conversie
ROUND		getal	INT	afronding
FLOOR		getal	INT	afronding naar beneden
CEIL		getal	INT	afronding naar boven
MIN		2×getal	getal	kleinste van 2 waarden
MAX		2×getal	getal	grootste van 2 waarden
ABS		getal	getal	absolute waarde
CONSTRAIN		3×getal	getal	$\min(\max(x, y), z)$
MAP		5×getal	getal	$(a - b)(e - d)/(c - b) + d$
POW		2×getal	getal	x^y
SQ		getal	getal	x^2
SQRT		getal	getal	$\sqrt{x}$
DELAY		getal		`delay(x)`
DELAYUNTIL		getal	getal	verlaag PC met 1 en push x als $x >$ `(int) millis()`, anders niets
MILLIS			INT	`(int) millis()`
PINMODE		2×getal		`pinMode(x, y)`
ANALOGREAD		getal	INT	`analogRead(x)`
ANALOGWRITE		2×getal		`analogWrite(x, y)`
DIGITALREAD		getal	CHAR	`digitalRead(x)`
DIGITALWRITE		2×getal		`digitalWrite(x, y)`
PRINT		waarde		output naar console
PRINTLN		waarde		output naar console met newline
OPEN		STRING		zet FP naar begin van bestand
CLOSE				doet niets
WRITE		waarde		schrijf naar file, FP wordt opgehoogd
READINT			INT	lees van file, FP wordt opgehoogd
READCHAR			CHAR	lees van file, FP wordt opgehoogd
READFLOAT			FLOAT	lees van file, FP wordt opgehoogd
READSTRING			STRING	lees van file, FP wordt opgehoogd
IF	1 byte	getal	getal	spring aantal bytes verder als $x = 0$; getal blijft staan
ELSE	1 byte	getal	getal	spring aantal bytes verder als $x \neq 0$; getal blijft staan
ENDIF		getal		verwijdert getal van stack
WHILE	2 bytes	getal	byte	spring (byte1 + 1) verder als $x = 0$, push anders (byte1 + byte2 + 4)
ENDWHILE		byte		spring aantal bytes terug
LOOP				zet PC in loop-register
ENDLOOP				maak PC gelijk aan loop-register
STOP				wis alle variabelen en beëindig proces
FORK		STRING	INT	start nieuw proces; push proces-ID
WAITUNTILDONE		getal	getal	verlaag PC met 1 en push x als proces x niet TERMINATED, anders niets