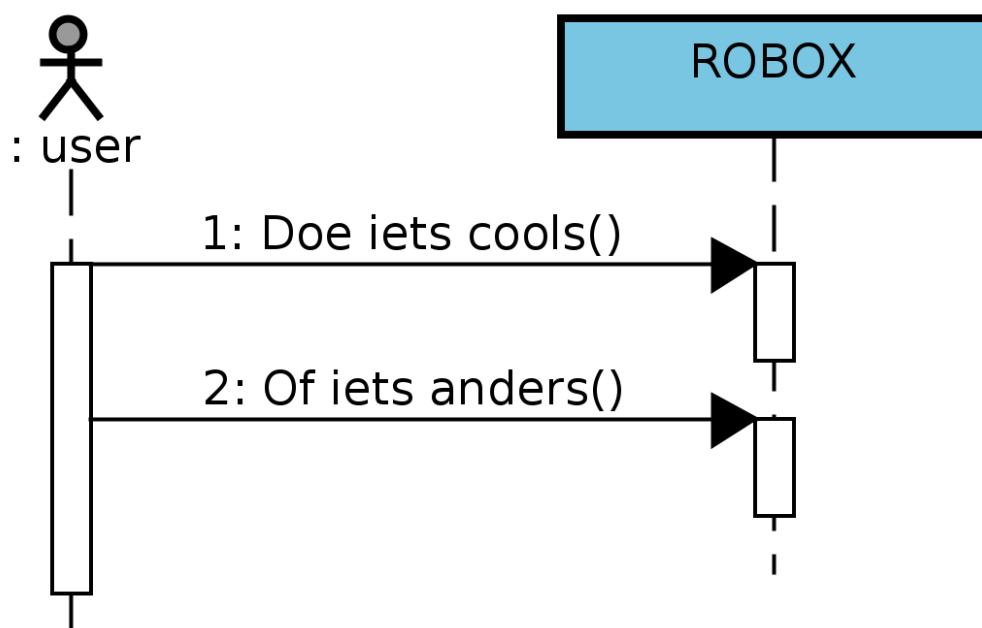


# ROBOX - Firmware

## Software design description



|                       |                                     |                         |                                 |
|-----------------------|-------------------------------------|-------------------------|---------------------------------|
| <b>Auteur</b>         | Luke van Luijn                      | <b>Minor</b>            | Digital Media Productions (DMP) |
| <b>Student nummer</b> | 587478                              | <b>Docentbegeleider</b> | Mario de Vries                  |
| <b>Opleiding</b>      | HBO-ICT                             | <b>Plaats</b>           | Nijmegen                        |
| <b>Profiel</b>        | Embedded Software Development (ESD) | <b>Datum</b>            | 27-05-2022                      |
| <b>Studiejaar</b>     | Jaar 3                              | <b>Versie</b>           | 1.0                             |

# Inhoudsopgaven

---

- 1 [Termen](#)
- 2 [Introductie](#)
  - 2.1 [Doel en domein](#)
  - 2.2 [Doelgroep](#)
  - 2.3 [Doel van het document](#)
- 3 [Architectural overview](#)
- 4 [Detailed design description](#)
  - 4.1 [Package - Communication](#)
    - 4.1.1 [Ontwerp keuzes](#)
  - 4.2 [Package - State machine](#)
    - 4.2.1 [Ontwerp keuzes](#)
  - 4.3 [Package - Device](#)
    - 4.3.1 [Ontwerp keuzes](#)
  - 4.4 [Package - Utils](#)
    - 4.4.1 [Ontwerp keuzes](#)
  - 4.5 [Package - Base](#)
    - 4.5.1 [Ontwerp keuzes](#)
- 5 [Literatuurlijst](#)

# 1. Termen

| Index | Term   | Beschrijving  |
|-------|--|---|
| 00    | <b>ROBOX/ Robot</b>                              | Met deze term wordt de fysieke robot ofwel het apparaat bedoelt.  |
| 01    | <b>UML</b>                                       | <i>Unified modeling language</i> De taal die gebruikt wordt voor het modelleren van een software systeem.   |
| 02    | <b>SRS</b>                                       | <i>Software requirements specification</i> Het ontwerp document van de geschreven software.   |
| 03    | <b>Package/<br/>Namespace</b>                    | Een groepering van software componenten die een soortgelijk doel nastreven.   |
| 04    | <b>Coupling</b>                                  | De graad van samenhang tussen verschillende componenten in een systeem (Wikipedia contributors, 2022a).   |
| 05    | <b>Overkoepelende<br/>systeem/<br/>gebruiker</b> | Met deze term wordt de grafische user interface bedoelt.  |
| 06    | <b>Seriële bus</b>                               | Een veelgebruikt, maar ouderwets, protocol waarmee verschillende apparaten, bijvoorbeeld via USB, data kunnen uitwisselen (Wikipedia-bijdragers, 2022).   |
| 07    | <b>Parsen</b>                                    | Het uitlezen van een <a href="#">ASCII</a> string, om hier vervolgens de correcte waardes uit te halen.   |
| 08    | <b>Singleton<br/>pattern</b>                     | Een design pattern die het mogelijk maakt om precies een instantie van een klasse te hebben (Sourcemaking, z.d.-a).   |
| 09    | <b>Facade pattern</b>                            | Een design pattern die complexe (sub) systemen verbergt en de functionaliteit toegankelijk maakt door middel van een 'deur'-class (Sourcemaking, z.d.-b).   |
| 10    | <b>State pattern</b>                             | Een design pattern die het mogelijk maakt op een elegante manier verschillende functionaliteiten uit te voeren op basis van verschillende inputs (Sourcemaking, z.d.-c).  |
| 11    | <b>Homing</b>                                    | De robot weet niet in welke positie de stepper motoren zich bevinden. Door een homing sequence uit te voeren weet de robot waar de motoren zich precies bevinden. Na een homing sequence kunnen er dus exacte bewegingen gedaan worden. |
| 12    | <b>Blokkeren</b>                                 | Het ophouden van de algemene 'flow' van de applicatie. Als iets blokkerend is staat verder alles stil tot de blokkade is verholpen.   |
| 13    | <b>Severity</b>                                  | De rangschikking van de log berichten. ERROR > WARNING > INFO > DEBUG.  |

## 2. Introductie

---

In dit document zal het ontwerp van de firmware beschreven worden. Hoe deze applicatie in essentie uitgewerkt is.

### 2.1. Doel en domein

Dit project, ROBOX, dient als een initiële opzet voor een groeiend project. De firmware zal in de toekomst verder uitgebreid en verbeterd worden of door de ontwikkelaar zelf of door externe belangstellende. Het doel van de huidige iteratie is om een goed werkend product te ontwikkelen dat de basis functionaliteit implementeert.

### 2.2. Doelgroep

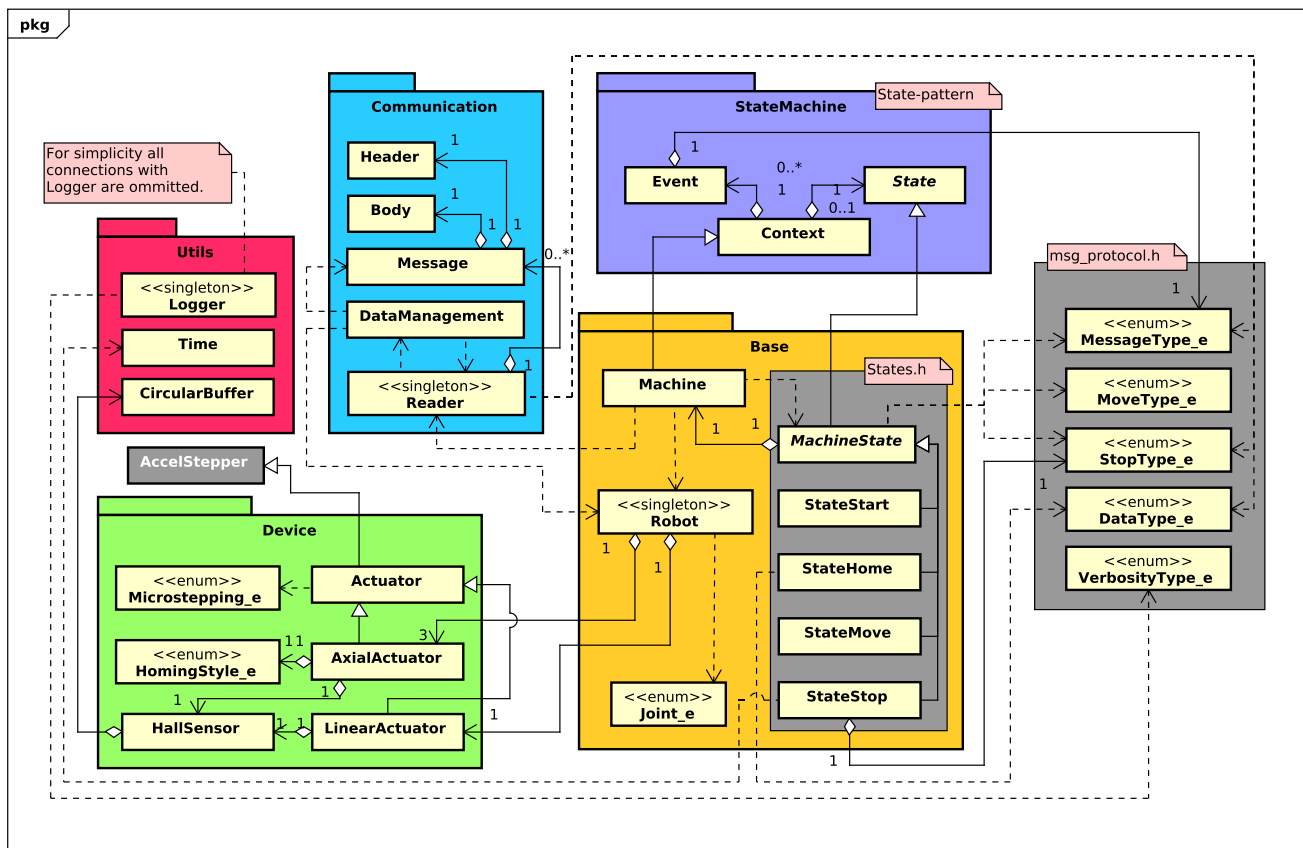
Het Software design description document is geschreven voor de projectbegeleiding en eventueel andere (externe) belangstellende. In dit document wordt aangenomen dat de lezer een basis kennis bevat van UML en softwareontwikkeling. Ook wordt er vanuit gegaan dat de lezer kennis heeft van het [SRS](#) document gemaakt voor de firmware.

### 2.3. Doel van het document

Dit document is opgezet om een duidelijk beeld te creëren van de werking van het product. In dit document worden de verschillende componenten toegelicht aan de hand van class diagrams en eventueel sequence diagrams.

### 3. Architectural overview

De firmware applicatie is onderverdeeld in meerdere *packages*, ook wel *namespaces*. Het algemene overzicht van de gehele applicatie is weergegeven aan de hand van een design klasse diagram (zie onderstaand). Vervolgens is er per package een klasse diagram opgesteld met een bijhorende beschrijving, deze volgen in de onderstaande onderdelen.



**Diagram 1 - Design class diagram - Firmware applicatie**

Zoals te zien in het bovenstaande diagram is de applicatie geschreven met een minimale *coupling* in het achterhoofd. Deze minimale coupling is gerealiseerd door waar mogelijk gebruik te maken van een *facade pattern*. Een voorbeeld van dit design pattern is te zien bij de klasse `Reader` en `Robot`, beide deze klasse fungeren als deur naar een dieper complex systeem. Hierover is meer te lezen in de onderstaande onderdelen.

## 4. Detailed design description

### 4.1. Package - Communication

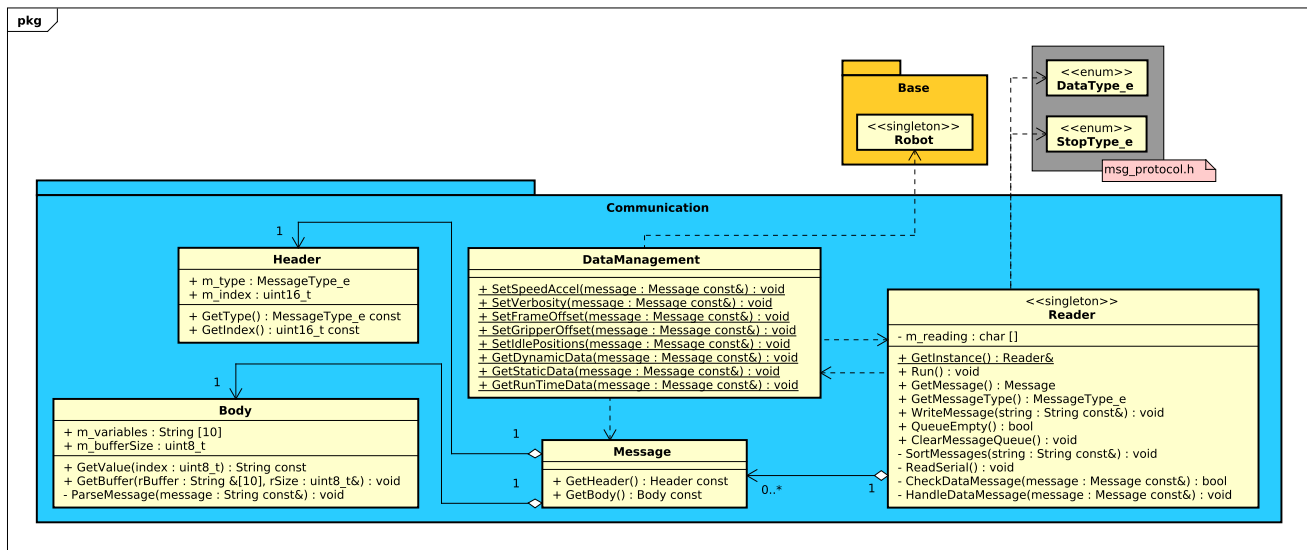


Diagram 2 - Class diagram - Communication package

De package **Communication** is verantwoordelijk voor de communicatie met het *overkoepelende systeem*. Het regelt beide het schrijven en lezen over de *seriële bus*. De communicatie is gebaseerd op de klasse **Message**, deze klasse is in essentie een buffer voor een inkomend bericht. Een message bestaat uit een **Header** en een **Body**. De Header bevat het type bericht en de variant van dit type (zie *message protocol*). De body bevat eventuele parameters die het bericht bevat. De klasse **DataManager** is verantwoordelijk voor het correct *parsen* en opstellen van de berichten. Deze functionaliteit is gebaseerd op het *message protocol*.

De klasse **Reader** is de klasse die daadwerkelijk de inkomende berichten leest. Omdat de seriële bus maar op een enkele locatie tegelijk aangesproken kan worden is er bij deze klasse gebruik gemaakt van een *singleton pattern*. Deze klasse kijkt continue naar de seriële bus, als er een bericht beschikbaar is zal deze gecontroleerd worden op basis van het *message protocol* en, indien goedgekeurd, gesorteerd worden in de messageQueue. Tijdens het sorteren zal er gekeken worden naar de header van de message. Als blijkt dat de header van de message een datatype is (DataType\_e) zal dit bericht direct afgehandeld worden door een response te creëren. Als het een E-Stop bericht te zijn zullen de motoren direct gedeactiveerd worden en wordt de gehele messageQueue geleegd zodat de robot stopt met bewegen en ook niet meer verder gaat. Als dit beide niet het geval is zal het bericht aan de messageQueue worden toegevoegd. Wanneer de state machine gereed is om een beweging te maken zal de messageQueue worden afgehandeld.

### 4.1.1. Ontwerp keuzes

|                     |  |
|---------------------|--|
| <b>Probleem</b>     | De seriële bus mag maar door een instantie tegelijk aangesproken worden.   |
| <b>Besluit</b>      | Er is voor een singleton pattern gekozen.  |
| <b>Alternatieve</b> | Handmatig bijhouden welke instantie wanneer de seriele bus aanspreekt.   |
| <b>Argumenten</b>   | Een singleton klasse heeft als eigenschap dat er altijd precies 1 instantie, dezelfde, actief is in de applicatie. |

**Tabel 1** - *Communication - ontwerp keuze 1*

|                     |   |
|---------------------|---|
| <b>Probleem</b>     | Hoe verzeker je dat de applicatie alle inkomende berichten altijd op dezelfde manier verwerkt ( <i>parsed</i> )?  |
| <b>Besluit</b>      | Het opstellen van een <b>Message</b> klasse. De klasse kan maar op een manier aangemaakt worden en is dus naast de inhoud altijd hetzelfde.   |
| <b>Alternatieve</b> | Het individueel parsen van de inkomende berichten waar nodig.   |
| <b>Argumenten</b>   | Door een <b>Message</b> te maken is er een gestandaardiseerde methode om berichten te maken en uit te lezen. Dit maakt het voor de huidige en toekomstige klasse makkelijk om om te gaan met de seriële communicatie. |

**Tabel 2** - *Communication - ontwerp keuze 2*

|                     |   |
|---------------------|---|
| <b>Probleem</b>     | Hoe zorg je dat de berichten verstuurd tussen de applicaties altijd op dezelfde manier opgesteld zijn?  |
| <b>Besluit</b>      | Het opstellen van een bestand message protocol ( <a href="#">msg_protocol.h</a> ), en dit bestand vervolgens voor beide applicaties gebruiken.                                      |
| <b>Alternatieve</b> | Erg goed opletten dat de berichten altijd volgens hetzelfde patroon verlopen.   |
| <b>Argumenten</b>   | Door een protocol op te stellen is het makkelijker om een correct geformuleerd bericht op te zetten. Het uitlezen van berichten is hierdoor ook veel minder ' <i>error-prone</i> '. |

**Tabel 3** - *Communication - ontwerp keuze 3*

## 4.2. Package - State machine

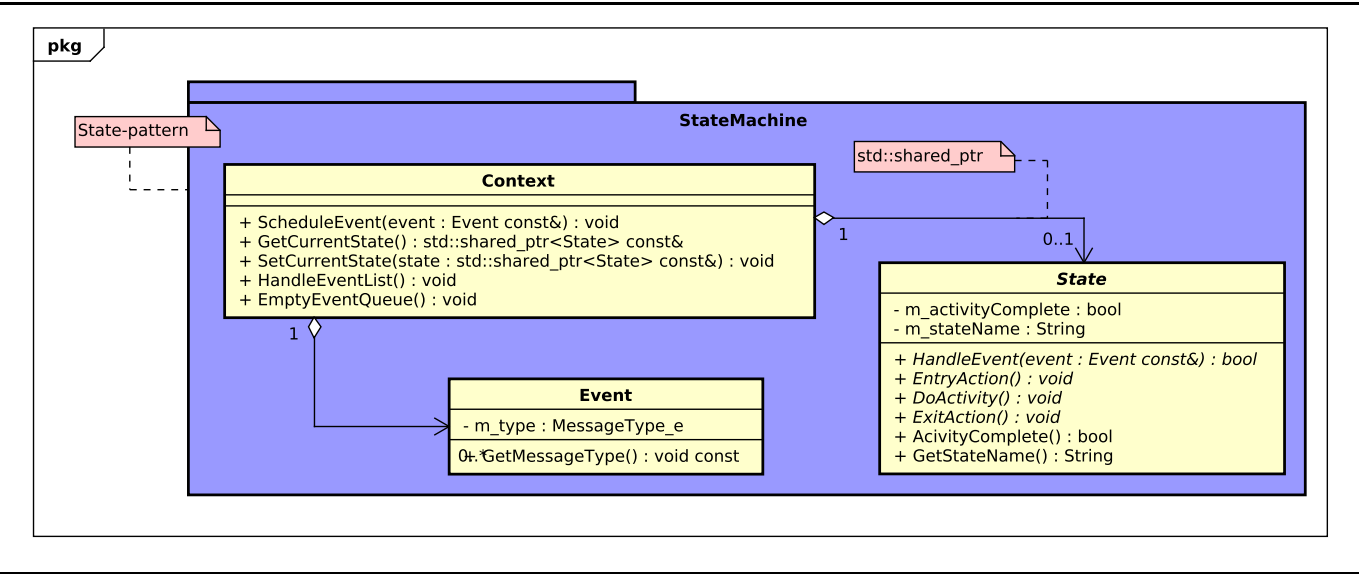


Diagram 3 - Class diagram - State machine package

De package **State Machine** is verantwoordelijk voor de opzet en uitwerking van de statemachine. Deze package is een uitwerking van de *state pattern*. Door gebruik te maken van een state machine is het makkelijk om op basis van een bepaalde input (een serieel bericht) een bepaalde functionaliteit uit te voeren.

De klasse **State** is een abstracte uitwerking van een state. Het is de bedoeling dat deze in de daadwerkelijke applicatie uitgewerkt wordt (zie Package - Base). De klasse **Event** fungeert als een trigger. Aan de hand van deze klasse kunnen verschillende transities getriggerd worden binnen de state machine. Tot slot de klasse **Context**. De Context klasse regelt de samenhang tussen state en event. Deze klasse bevat een lijst met events die een voor een afgehandeld worden. Ook heeft deze klasse een instantie van de huidige state. Context is ook verantwoordelijk voor het uitvoeren van de transities tussen states.

### 4.2.1. Ontwerp keuzes

|              |  |
|--------------|--|
| Probleem     | Hoe voer je verschillende functionaliteiten uit op basis van input?  |
| Besluit      | Het opstellen van een state machine, volgens het state pattern.  |
| Alternatieve | Een omvangrijke 'if else'-boom, switch case.   |
| Argumenten   | Een state machine is een elegante oplossing voor het probleem. Het levert de mogelijkheid om op een overzichtelijke manier de verschillende situaties (states) af te handelen. |

Tabel 4 - State machine - ontwerp keuze 1



### 4.3. Package - Device

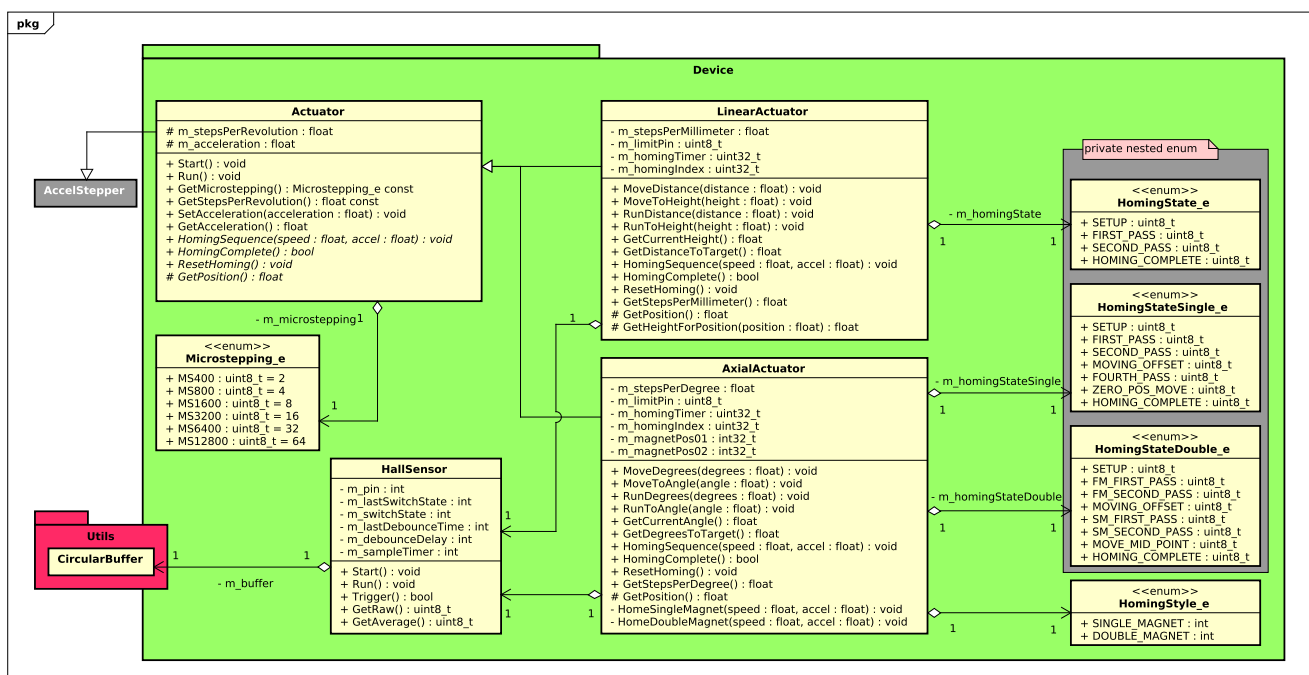


Diagram 4 - Class diagram - Device package

De package **Device** heeft als verantwoordelijkheid het direct aanspreken van de verschillende hardware componenten. De klasse **Actuator** is de overkoepelende actuator klasse die de algemene zaken regelt die voor beide onderliggende actuator classes hetzelfde zijn. **Linear** & **AxialActuator** zijn de daadwerkelijke actuators. De applicatie heeft drie axiale actuators en een lineaire actuator. De axiale actuator regelt beweging in graden en de lineaire actuator regelt beweging in millimeters. Beide deze uitwerkingen maken gebruik van een **HallSensor** instantie voor *homing*. De enumeraties **HomingState\_e**, **HomingStateSingle\_e** & **HomingStateDouble\_e** worden gebruikt voor de homing sequence van de actuators.

#### 4.3.1. Ontwerp keuzes

|                     |  |
|---------------------|--|
| <b>Probleem</b>     | Hoe kun je een homing sequence uitvoeren voor vier verschillende motoren zonder dat deze sequence het resterende programma <i>blokkeert</i> ?  |
| <b>Besluit</b>      | Het gebruik van verschillende enumeraties die de verschillende stadia van de sequence weergegeven zodat het voor de applicatie duidelijk is welke actie ondernomen moet worden.  |
| <b>Alternatieve</b> | Een homing sequence gebruiken die wel blokkeert.   |
| <b>Argumenten</b>   | Gezien de lineaire actuator vrij traag is en de axiale actuators ook niet bijster snel ontstond er een homing sequence van ongeveer twee minuten, in blokkerende toestand. Het was dus noodzakelijk om een niet-blokkerende homing sequence op te zetten zodat in ieder geval alle actuator tegelijkertijd de sequence konden uitvoeren. |

Tabel 5 - Device - ontwerp keuze 1

## 4.4. Package - Utils

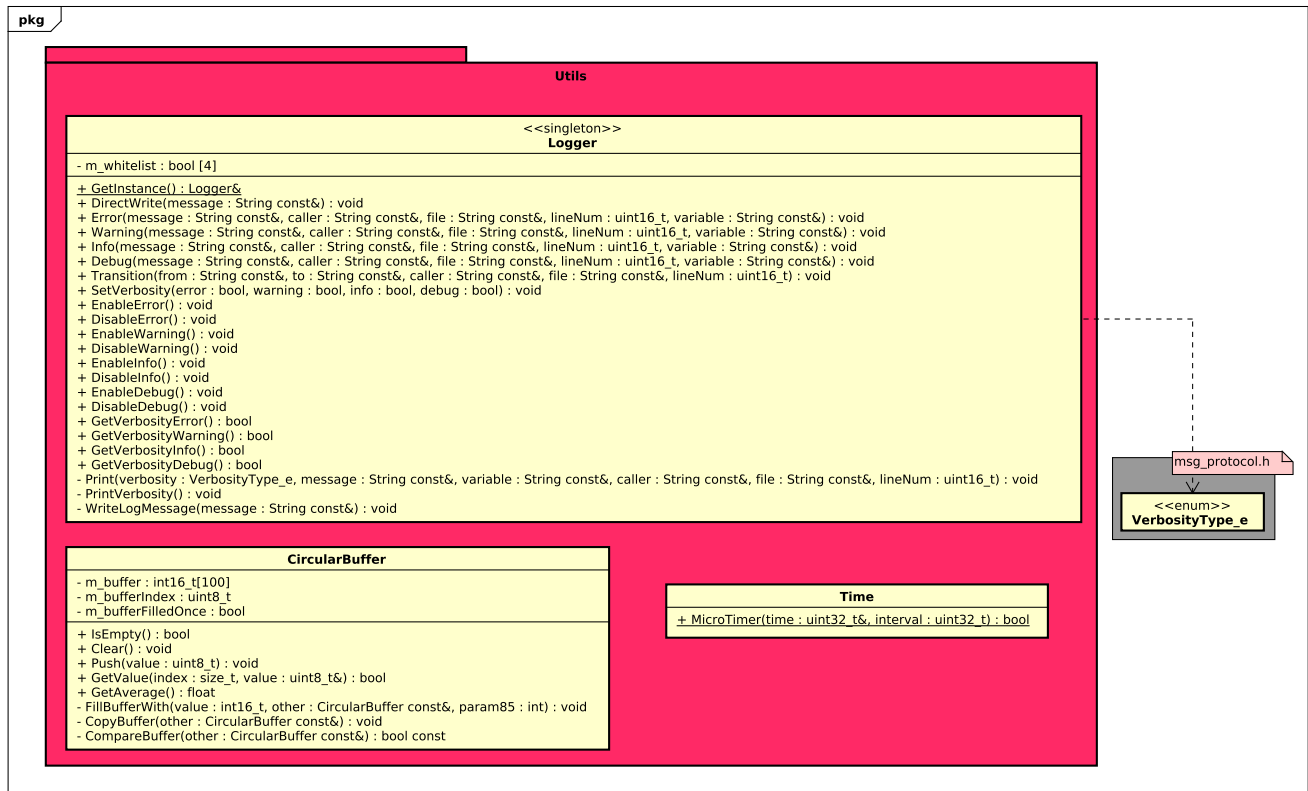


Diagram 5 - Class diagram - Utils

De **Utils** package is een verzameling van verschillende klasse die ieder een eigen functionaliteit bevatten. De klasse **Time** bestaat uit een enkele methode *MicroTimer*. De klasse **CircularBuffer** is een C++ implementatie van de *circulaire buffer* (Wikipedia contributors, 2022a) bedoelt voor het efficiënt opslaan en uitlezen van sensor waarde.

De **Logger** klasse heeft iets meer substantie dan de andere twee. De logger wordt door vrijwel de gehele applicatie gebruikt. De logger geeft de mogelijkheid om berichten naar de console te schrijven in een voorbepaald format. De Logger is de enige klasse die naar de seriële bus mag schrijven en is daarom, net zoals de reader, een singleton. Dit maakt het en makkelijk om vanaf meerdere locaties in de applicatie gebruik te maken van de logger en zorgt ervoor dat er nooit van meerdere locaties naar de seriële bus geschreven wordt. De logger houdt ook de *severity* van een bericht bij. Er zijn vier verschillende severities; ERROR, WARNING, INFO en DEBUG. Ieder van deze niveaus kan worden uit- en ingeschakeld. Naast de severity wordt ook de locatie van aanroepen meegenomen in het bericht. De methode, bestandsnaam en lijnnummer moet worden bijgehouden. Omdat het meegegeven van al deze data nogal veel typ-werk is er bij de logger ook gebruik gemaakt van verschillende *Macro's*, deze macro's zijn [hier](#) terug te vinden. Door gebruik te maken van deze macro's is het opstellen van een log-bericht een stuk simpeler.

Zonder macro      `Utils::Logger::GetInstance().Error("message",  
__func__, __FILE__, __LINE__, "vars");`

Met macro        `ERROR("message", "vars");`

Door log berichten te voorzien van deze verschillende data-elementen is het tijdens het afspelen van de applicatie vele malen makkelijker om bij te houden wat er precies gebeurt in de applicatie.

#### 4.4.1. Ontwerp keuzes

|                     |  |
|---------------------|--|
| <b>Probleem</b>     | Hoe kan op een efficiënte manier data opgeslagen en uitgelezen worden, verkregen van sensoren?   |
| <b>Besluit</b>      | Het opstellen van een circulaire buffer.   |
| <b>Alternatieve</b> | Grote arrays, of dynamische opties zoals vectoren.   |
| <b>Argumenten</b>   | Een circulaire buffer is erg efficiënt qua opslag en geheugen. Het levert altijd een relatief betrouwbaar gemiddelde van de huidige data en is makkelijk in gebruik. |

**Tabel 6** - *Utils - ontwerp keuze 1*

|                     |   |
|---------------------|---|
| <b>Probleem</b>     | Hoe verzeker je dat er maar een instantie tegelijkertijd naar de seriële bus schrijft.  |
| <b>Besluit</b>      | Er is wederom gekozen voor het gebruik van het singleton pattern.   |
| <b>Alternatieve</b> | Zeer precies bijhouden wanneer er geschreven wordt naar de seriële bus.   |
| <b>Argumenten</b>   | Zoals eerder vermeld, een singleton heeft maar een enkele instantie. Het is daarom een ideale keuze voor een probleem als deze. |

**Tabel 7** - *Utils - ontwerp keuze 2*

|                     |  |
|---------------------|--|
| <b>Probleem</b>     | Hoe kun je als gebruiker van de applicatie duidelijk zien wanneer, waar, wat gebeurt?  |
| <b>Besluit</b>      | Door tijd, methode, bestandsnaam en lijnnummer bij te houden van elke aanroep van een log bericht.   |
| <b>Alternatieve</b> | deze data handmatig in het bericht te verwerken.   |
| <b>Argumenten</b>   | De bovenstaande velden zijn ingebouwde macro's van C++ en zijn opgesteld precies om de reden die hier als probleem is opgesteld. Door deze data mee te geven aan elk log bericht is het meteen duidelijk waar er gekeken moet worden bij een onverwachte situatie. |

**Tabel 8** - *Utils - ontwerp keuze 3*

|                     |   |
|---------------------|---|
| <b>Probleem</b>     | Hoe kun je op efficiënte manier log berichten opstellen met toch de adequate data onderdelen.                               |
| <b>Besluit</b>      | Het opstellen van verschillende macro's die al deze datavelden automatisch aanroepen.                                       |
| <b>Alternatieve</b> | Het handmatig meegeven van deze velden.   |
| <b>Argumenten</b>   | Zoals verteld zijn de velden ingebouwde C++ macro's en zijn daarom ideaal om gebruikt te worden voor een situatie als deze. |

Tabel 9 - Utils - ontwerp keuze 4

## 4.5. Package - Base

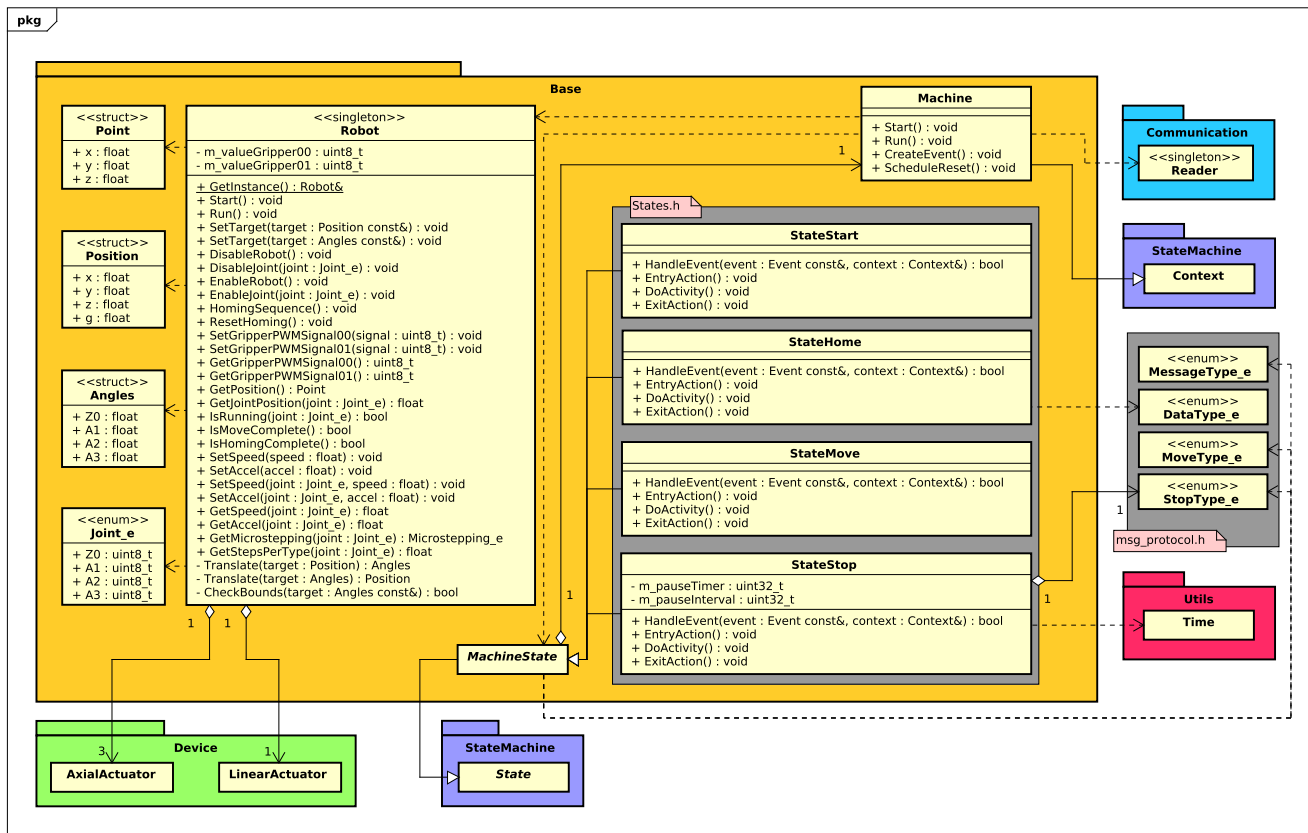


Diagram 6 - Class diagram - Base

De **Base** package is het centrale punt van de applicatie. In deze package is de samenhang tussen de package communication, StateMachine en device geregeld. De base package is opgedeeld in twee hoofd onderdelen.

Aan de ene kant is de **Robot** klasse. Deze klasse is verantwoordelijk voor alles hardware. In deze klasse worden de verschillende actuatoren aangestuurd en gevalideerd. De Robot klasse is gemaakt volgens het singleton pattern, net zoals bij de seriële bus mag er altijd, maar precies, een instantie gebruik maken van de hardware. Verder wordt in de Robot klasse nog **inverse en forward kinematica** toegepast voor het vertalen van hoeken naar positie en van positie naar hoeken.

Aan de andere kant de state machine implementatie. De klasse **Machine** is een afgeleide van de klasse Context en draagt dezelfde verantwoordelijkheid. De klasse **MachineState** is een directe afgeleide van de State klasse, alle onderliggende state-klasse staan hieronder beschreven;

| Class             | Beschrijving   |
|-------------------|--|
| <b>StartState</b> | Start state is de staat waar de applicatie wacht op input van de gebruiker. In deze state zijn alle motoren uitgeschakeld.   |
| <b>HomeState</b>  | In deze state voert de applicatie de homing sequence uit voor de vier verschillende actuatoren. Tijdens deze state is het mogelijk om een E-Stop uit te voeren. Zodra de homing sequence voltooid is zal de flag: 'activityComplete' op true gezet worden. |

| Class     | Beschrijving   |
|-----------|--|
| MoveState | In de move state worden de motors direct aangesproken en zal de robot gaan bewegen. Zodra de beweging voltooid is zal de flag 'activityComplete' op true gezet worden. Ook tijdens deze state kan er een E-Stop uitgevoerd worden.   |
| StopState | In de stop state wordt het stoppen en pauzeren van de robot geregeld. Bij het pauzeren van de robot wordt er een timer aangezet, zodra de timer afloopt zal de activityComplete flag op true gezet worden. Bij het stoppen van de robot wordt de robot naar de <i>idle positions</i> bewogen. Zodra de robot aangekomen is op deze posities worden de motoren uitgeschakeld en de activityComplete flag op true gezet. |

Tabel 10 - State beschrijvingen

In het onderstaande state machine diagram is het verloop van de statemachine nogmaals afgebeeld.

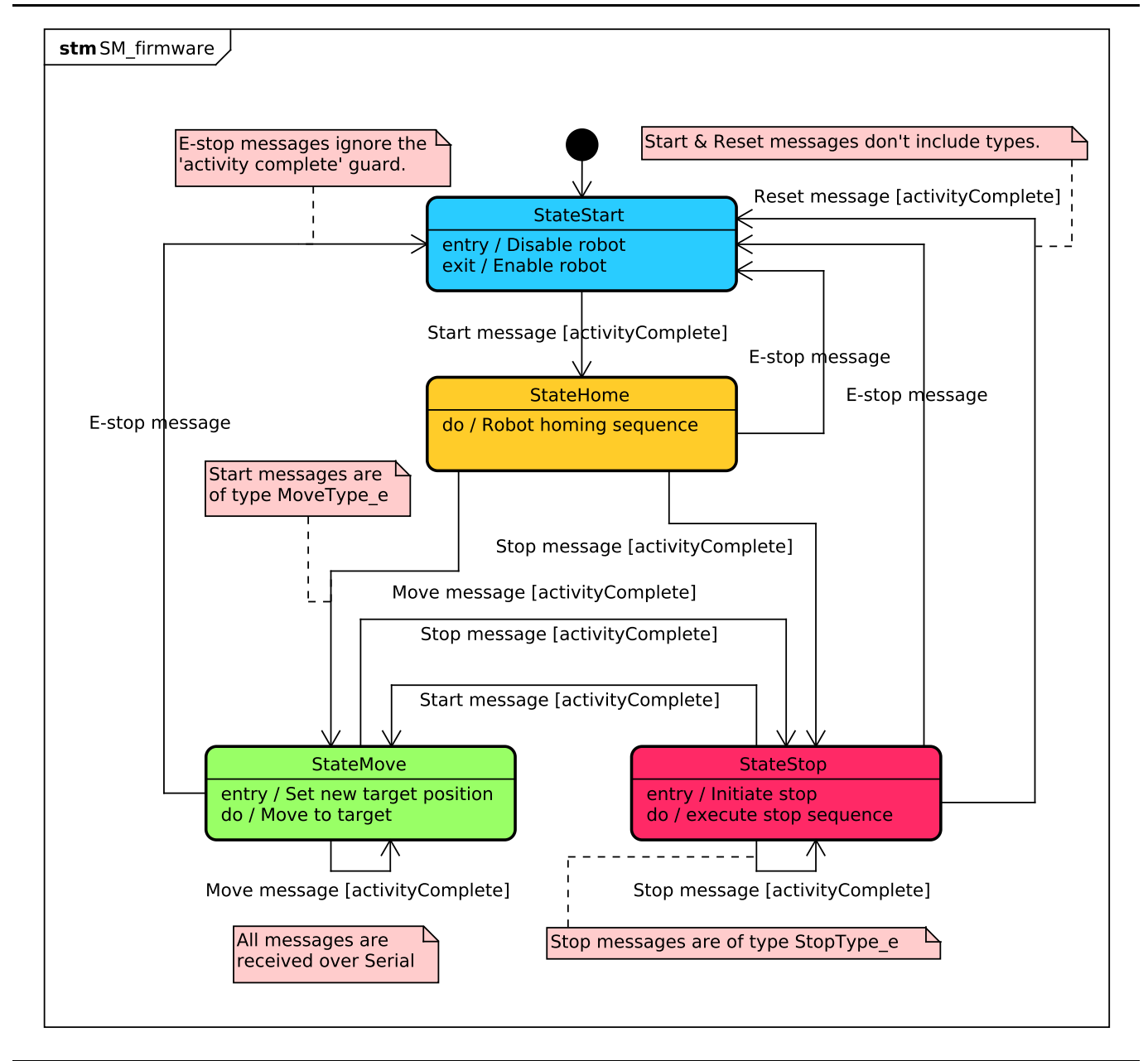


Diagram 7 - State machine diagram

### 4.5.1. Ontwerp keuzes

|                     |  |
|---------------------|--|
| <b>Probleem</b>     | Hoe verzek je dat er maar een instantie tegelijkertijd met de hardware componenten communiceert?   |
| <b>Besluit</b>      | Het opstellen van een singleton class.   |
| <b>Alternatieve</b> | Het zeer accuraat bijhouden wie er met de componenten praat.   |
| <b>Argumenten</b>   | Zoals meerdere malen verteld is een singleton uitermate geschikt als een restrictie voor het vanaf verschillende locaties praten met onderdelen. |

**Tabel 11** - Base - ontwerp keuze 1

|                     |  |
|---------------------|--|
| <b>Probleem</b>     | Hoe kun je vanuit hoeken voor iedere joint berekenen wat de uiteindelijke positie van de gripper zal zijn?                                   |
| <b>Besluit</b>      | Het toepassen van forwards kinematica.   |
| <b>Alternatieve</b> | geen.  |
| <b>Argumenten</b>   | Het toepassen van forwards kinematica levert de mogelijkheid om vanuit de eerder genoemde hoeken de eindpositie van de gripper te berekenen. |

**Tabel 12** - Base - ontwerp keuze 2

|                     |   |
|---------------------|---|
| <b>Probleem</b>     | Hoe kun je vanuit een eindpositie van de gripper bepalen wat de hoeken moeten zijn voor de rest van de robot?                                       |
| <b>Besluit</b>      | Get toepassen van inverse kinematica.   |
| <b>Alternatieve</b> | geen.   |
| <b>Argumenten</b>   | Het toepassen van inverse kinematica levert de mogelijkheid om vanuit de eerder genoemde positie de verschillende hoeken van de robot te berekenen. |

**Tabel 13** - Base - ontwerp keuze 3

## 5. Literatuurlijst

---

| Index | Source   |
|-------|--|
| 1     | Sourcemaking. (z.d.-a). Design Patterns and Refactoring. Singleton. Geraadpleegd op 27 mei 2022, van <a href="https://sourcemaking.com/design_patterns/singleton">https://sourcemaking.com/design_patterns/singleton</a>                                       |
| 2     | Sourcemaking. (z.d.-b). Design Patterns and Refactoring. Facade. Geraadpleegd op 27 mei 2022, van <a href="https://sourcemaking.com/design_patterns/facade">https://sourcemaking.com/design_patterns/facade</a>  |
| 3     | Sourcemaking. (z.d.-c). Design Patterns and Refactoring. State. Geraadpleegd op 27 mei 2022, van <a href="https://sourcemaking.com/design_patterns/state">https://sourcemaking.com/design_patterns/state</a>   |
| 4     | Wikipedia contributors. (2022a, mei 6). Coupling (computer programming). Wikipedia. Geraadpleegd op 27 mei 2022, van <a href="https://en.wikipedia.org/wiki/Coupling_(computer_programming)">https://en.wikipedia.org/wiki/Coupling_(computer_programming)</a> |
| 5     | Wikipedia contributors. (2022b, mei 23). ASCII. Wikipedia. Geraadpleegd op 27 mei 2022, van <a href="https://en.wikipedia.org/wiki/ASCII">https://en.wikipedia.org/wiki/ASCII</a>  |
| 6     | Wikipedia-bijdragers. (2022, 10 februari). Universal serial bus. Wikipedia. Geraadpleegd op 27 mei 2022, van <a href="https://nl.wikipedia.org/wiki/Universal_serial_bus">https://nl.wikipedia.org/wiki/Universal_serial_bus</a>                               |
| 7     | Wikipedia contributors. (2022a, april 26). Circular buffer. Wikipedia. Geraadpleegd op 27 mei 2022, van <a href="https://en.wikipedia.org/wiki/Circular_buffer">https://en.wikipedia.org/wiki/Circular_buffer</a>  |