Documentation of Boptcoeff and Path execution

Frederik Hartig & Jeppe Fjederholdt Nielsen

May 2020

Contents

1 Introduction

This document describes what the Boptcoeff is, and how to use it, it also describes the method of loading and executing the paths. The boptcoeff file should be included in the workcell file and should be named like the workcell-id or robot. The boptcoeff is an important tool for path optimization. The boptcoeff can be simulated by using the mpnguide from sandbox. To apply it to the right worckell you'll have to change the mntdatabase.txt file in the mpnguide folder. The contents of the mntdatabase.txt should be:

```
(github folder)/MRN-Software/(robot to simulate)/database
(workcellid)system.ini
```

For a more detailed user guide to mpnguide and manipulating the boptcoeff see section 3.
If a deeper explanation for boptcoeff is needed see section 2

For the robot to run the paths in the boptcoeff it must load and execute them with commands. For more on the correct way of loading and executing see section 4.
In general the *xyz* coordinate system is realted to robot, with **X** being the distance away from the gantry and the robot, **Y** being the height from the floor, **Z** being the gantry.

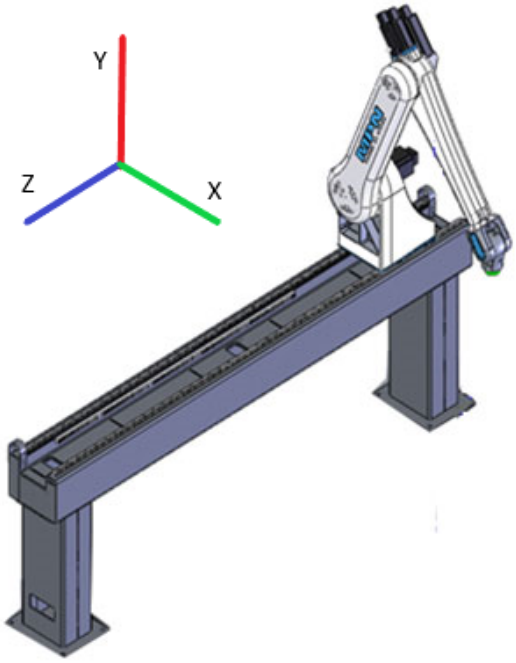


Figure 1: RoboStacker

1.1 Frames

In order to be able to move from one place to another the program has a valuetype called **Frame**. Frame is used to set a certain start or end position for different paths. A Frame could be *palletB*, used for palletizing. The frame has six parameters:

```
Frame=palletB x=0.00 y=0.00 z=0.00 v=0.00 w=0.00 u=0.00
```

This will make a frame in orgio called palletB.

2 Boptcoeff

2.1 Description

The boptcoeff stands for *"Bane Optimerings Coefficient"* or *"Path Optimizing Coefficient"* in english. It's a tool to make the robots path more efficient and different options for trajectory and pathing, explained in section 2.3 and section 2.8.

2.2 Boptcoeff file

First line is the Version of boptcoeff which is always "1".
Second line is the workcell-id/robot.

```
BOPTCOEFFVERSION 1
350
```

To make a path, the following sections will explain the contents. and a general guide will be in section **??**.

2.3 Path

A path is the route that the robot has to take, to get from one frame to another. The path always need to have these things:

- Mode
- Pathtype
- Pathname
- Viatype
- Via point

2.4 Mode and pathtype

Every paths taken by the robot must be initiated with a mode and **pathtype**, these to parameters make an unique id for the path and tells the robot which boptcoeff path to use. The mode is a number that makes the path unique because two different paths can have the same **pathtype** if the operation should be identical but the trajectory should be different. An unique mode and type could look like

```
mode 16 pathtype 20 :
```

The **pathtype** is read by the Trio, and the Trio decides the actions of the tool across the path. The Trio code will not be explained in this document, but look into the code and find **pathtype** to see the actions related to the number.

2.5 PathName

A path must have a name, but the name in the boptcoeff file does not have to be the same as the name in the statemachine where it is loaded. The correct way to name a path is **starteFrame_endFrame** e.g. **home_palletA**
To see how a path is being handled by a statemachine go to section 4.

2.6 Def (for simulation)

The **def** setting is for defining frames, items, and patterns. Patterns are not generally used in the robot program. The posibilities with **def** are:

- def.fromFrame
- def.toFrame
- def.fromItem
- def.toItem

- def.fromPattern
- def.toPattern

Section 4 will explain how the **fromFrame** and **toFrame** is assigned. The general way of using **def** in the boptcoeff file is:

```
def.fromFrame home
def.toFrame palletA
def.toItem C18
```

2.7 Viatype

To determine which via points the path should use, the **viatype** needs to be set, shown in table 1

viatype	**via points used**
NONE	Do not use any via points
START	Only use via point 1
END	Only use via point 2
BOTH	Use via point 1 and 2

Table 1: Viatype

A viatype would be set as so:

```
viatype BOTH
```

2.8 Via points

Via points is a point the robot must reach before going to the next/end point in it's path. the points parameter are in the x,y,z coordinates. The path can have up to two via points. **via1pos** and **via2pos**. The via points are relative to the frames, meaning the **via1pos**'s xyz is relative to the start frames xyz, and the **via2pos**'s xyz is relative to end frames xyz.
The via point is set with x,y and z individually, but can also Meaning if you for example want the **via1pos** to be $1000mm = 1m$ over the start frame you can write:

```
via1pos.y c=500
via2pos.y c=1000
```

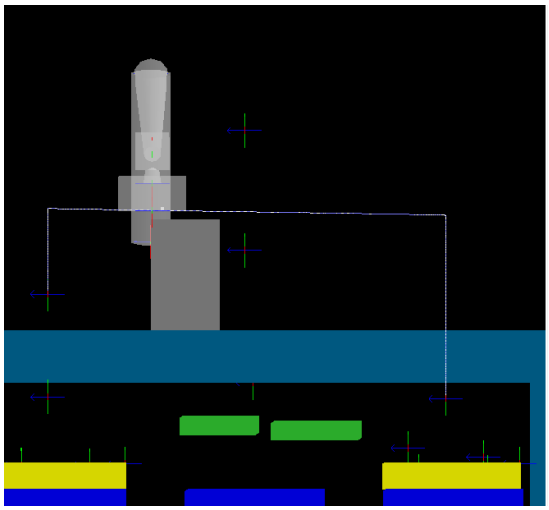


Figure 2: Simple path with only y parameters on the via point

Like frames a Via point has six different values, they are by default 0 but can be set:

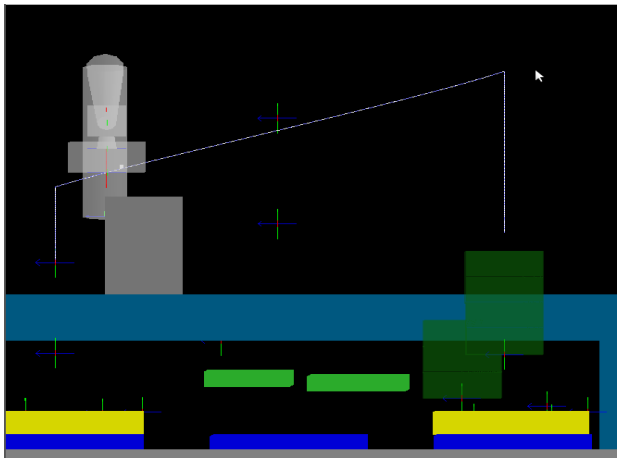
Value	Robot Relation
viaXpos.x	X coordinate for the TCP distance from the gantry
viaXpos.y	Y coordinate for the TCP height
viaXpos.z	Z coordinate for the TCP along the gantry
viaXpos.v	The tilt of the d-axis(tool) in degrees (90=Horizontal 0=Vertical)
viaXpos.w	The angle of the e-axis(tool) in degrees (90=parallel with the gantry)
viaXpos.u	Not used in the current program

Table 2: Via point values

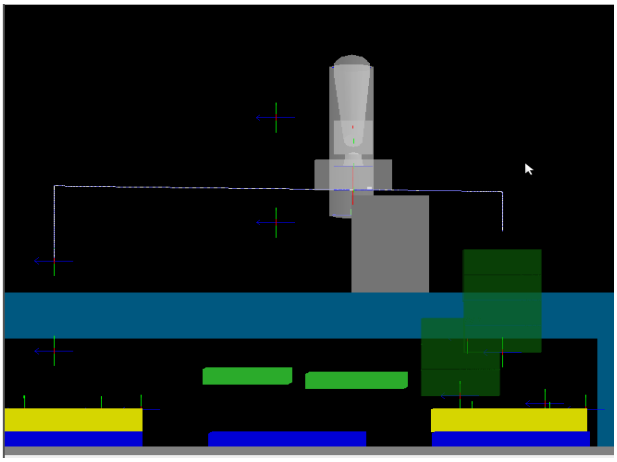
Via point can be complex by adding some variables. The variables are letters with a number ranging from 0 to 1, see Table 3. The number is calculated by where the de-/palletizing frame position is compared to a minimum and maximum value.
E.g. if the path is going from pickup to the palletizing frame, and the path parameters are a minheight of 200mm and a maxheight of 2200mm. It's demanded that the **via2pos** must always be at the same height of 2400mm. The via point should be:

```
via2pos.y c=200 p=2000
```

When the palletizing frame is at maxheight 2200mm, $p = 0$ and the **via2pos** is $c = 200mm$ above the frame. When the palletizing frame is at minheight 200mm, $p = 1$ and the **via2pos** is $c = 200mm + p = 2000mm => 2200mm$ above the frame. The variables are connected to the either, toFrame, fromFrame or both frames, and the frames position on x,y or z.



(a) via2pos.y c=1000



(b) via2pos.y c=0 p=1000

Figure 3: The difference between using frame variables and not

The example at figure 3 is important to get the path to work properly The variable can be seen in Table 3:

Var	Frame	Range of value
g	toFrame	$0 = minheight$ to $1 = maxheight$
p	toFrame	$1 = minheight$ to $0 = maxheight$
b	fromFrame	$0 = minheight$ to $1 = maxheight$
f	fromFrame	$1 = minheight$ to $0 = maxheight$
l	frame distance difference	$0 = minlength$ to $1 = maxlength$
h	frame height difference	$0 = minheight$ to $1 = maxheight$
r	rotational difference	1 when the frames is 360 degrees apart on the E-axis
t	tilt difference	1 when the frames is 180 degrees apart on the D-axis

Table 3: Via point frame variables

For a somewhat visual representation of the via point frame variables see figure 4

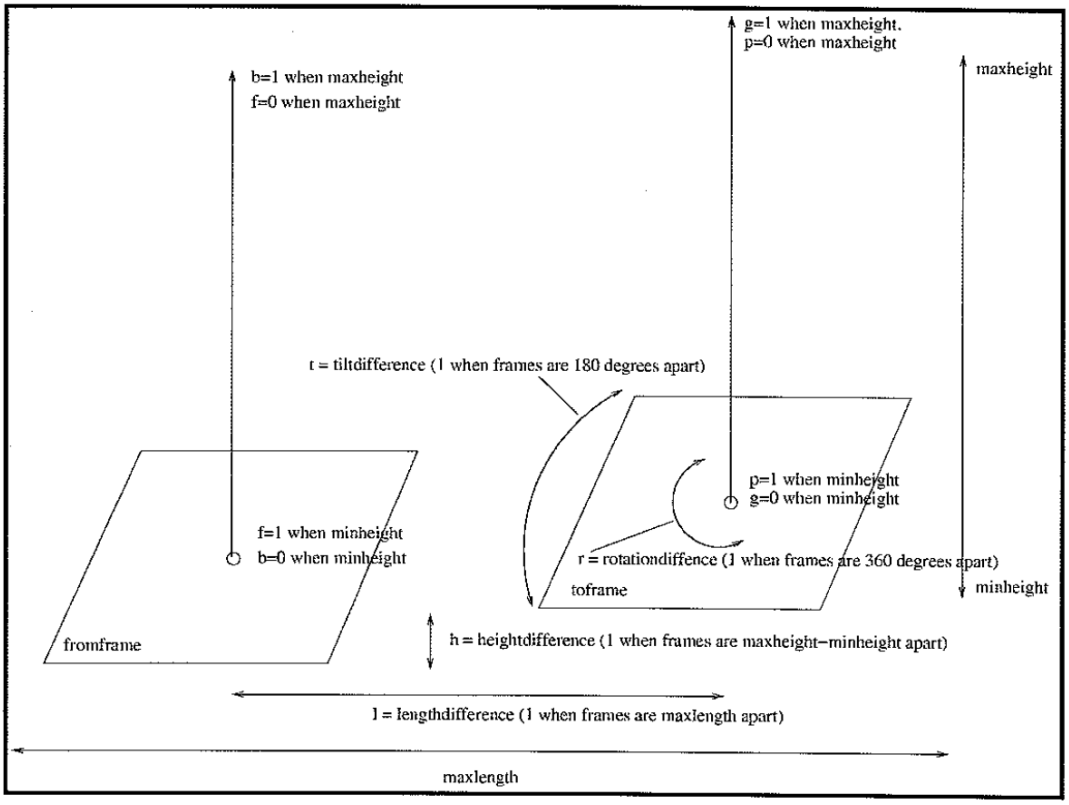


Figure 4: Frame illustration

By using one of the letters the path optimization function becomes complex for the via point, for example:

```
via1pos.y c=1000 f=100 bb= 200
via1pos.z c=200 l= 100
via1pos.x c=0
via2pos.y c=500 g=100 ggg= 200
via2pos.z c=100
via2pos.x c= 100
```

INSERT DECRIPTION OF POS.V AND POS.W FOR ROTATING THE TCP

If the path shouldn't move in one direction simply don't set it. The coordinates are by default the same as their start/end frame.

As described before, the via point is a stop point through the route. to make the stop less significant, velocity can be added, for example:

```
via1vel.y c=100
via1vel.z c=100
```

The **via1vel** adds velocity to **via1pos** making it exit the via point faster and giving the path a more curve like trajectory. Using mpnguide simulation, the velocity is viewed as a vector arrow.

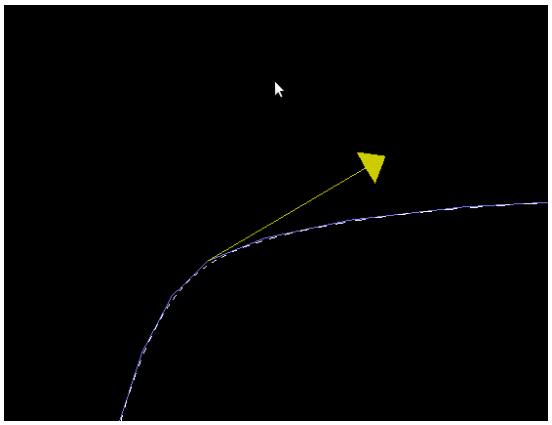


Figure 5: Velcoity arrow

2.9 Macro

Macros are added to the start or the end of the path. A macro will do a permanent movement from or to the assigned frame. An example of a macro can be:

```
macro1 type=FASTUPLINE length=50 time=0.3
```

The macro syntax has four important parameters:

- Macro1 - is related to the start frame
- Type - describes the function, a fastupline moves the TCP up fast.
- Length - is how long the macro should move.
- Time - is how fast the macro should move.

The different macro types in Table 4.

Macro types	**movement**
UPLINE	TCP moves up the Y coordinate by the distance of length
FASTUPLINE	TCP moves fast up the Y coordinate by the distance of length
DOWNLINE	TCP moves down the Y coordinate by the distance of length
FASTDOWNLINE	TCP moves fast down the Y coordinate by the distance of length
SPIRAL	*INSERT TEST RESULTS HERE*

Table 4: Macro types

Some important things to consider is that the length and time has to match or else the macro will be too fast, resulting in a corrupted path. The length should also match the via point to get a good result. A time too small and the velocity will make the Robot go higher then you want or make the simulation bug out. A length higher then your via point will make it go above the point and down.

2.10 Max velocity and max acceleration

Max velocity and max acceleration can be set for each path, the velocity and acceleration is set on all axis individually. For example:

```
mv a=3.7000 b=3.1100 c=3.2400 d=2.6200 e=5.2360
ma a=4.5000 b=4.2000 c=5.0000 d=6.0000 e=6.0000
```

It is recommended to have some standard value to all the paths. Some velocity's may look fine at the start, but running the program on 100%, if the mv or ma is to high, can make the encoder skip and throw a following error.

2.11 Boptcoeff Path example

An example of a full path can be:

```
mode 16 pathtype 21 :
pathName pickup_palletB
def.fromFrame palletB0
def.toFrame conveyorB
viatype BOTH
via1pos.y c=50 f=1000
via1pos.z c=100
via1vel.y c=200
via1vel.z c=120
via2pos.y c=540
via2pos.z c=-100 r=-200
via2pos.w r=-360
via2vel.y c=-100
via2vel.z c=100 r=-50
macro1 type=FASTUPLINE length=50 time=0.3
macro2 type=FASTDOWNLINE length=50 time=0.3
mv a=3.0 b=3.11 c=3.24 d=2.62 e=5.236
ma a=6.5 b=4.2 c=5.0 d=6.0 e=6.0
params tcpidx=0 blendtype=0 maxheight=1635 minheight=635 maxlength=3500 samplerate=15 timefactor=1
```

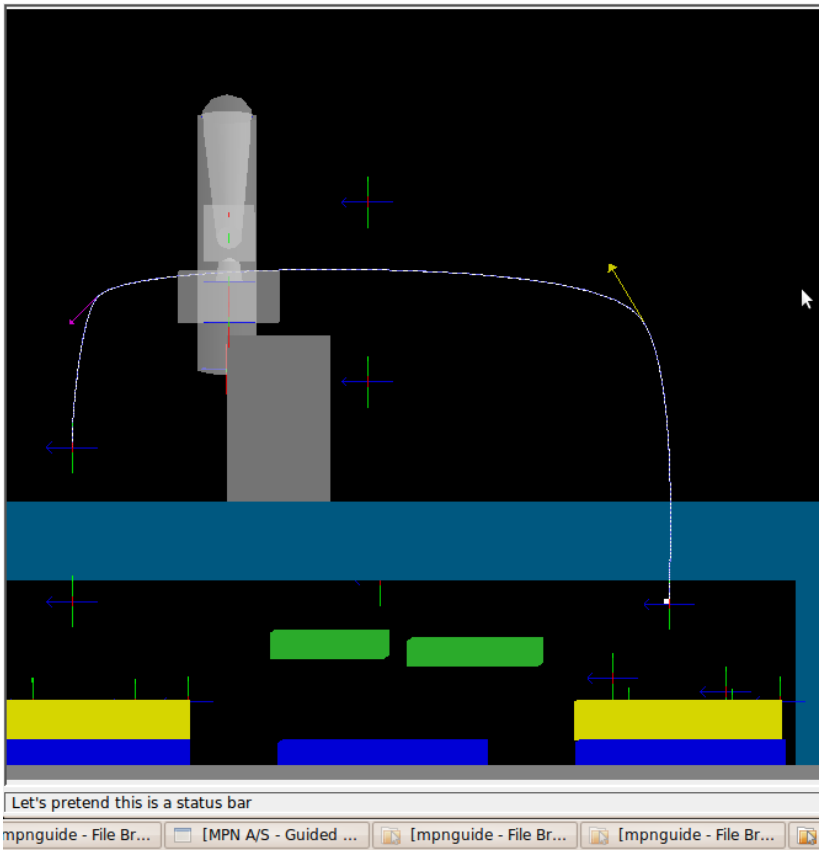


Figure 6: Path example

The path is designed to pickup an item in a variable height and place it in a constant height. Because of the variable height on the first frame the "f" variable is used. To get some round and smooth via points, velocity is added and the via points z-axis gets a small offset. Sometimes the item have to rotate, but rotation shall only happen after via2pos. that's why **via2pos.w r=-360**, the Velocity needs to be smaller when the path have to rotate that's why **via2vel.z c=100 r=-50**

3 Using mpnguide to simulate

3.1 introduction to mpnguide

Mpnguide is a 3D simulation software for offline programming of the robots paths and system. The program uses the system.ini file from the robot which should be simulated, so same as the robostacker program, the location of the ini file is read from the mntdatabase.txt file. To change the simulation of the system, accessory files can be made or modified, more on accessory files in section 3.2. mpnguide has some easy usable functions for path optimizing, see section 3.3. Mpnguide uses the boptcoeff file include in the workcell of the robot, to simulate the paths. In the boptcoeff it is recommended to use the **def** command from section 2.6. to define fromFrame, toFrame, fromItem, toItem, or fromPattern and toPattern. For an accurate simulation, the defined frames should be the same as the frames of the given path in RobotProgram.

3.2 Accessory

The accessory files in the database is used to simulate conveyor belts, pallets, pallet segments and more. The content of an accessory file is:

```
VERSION 1
[NameOfAccessory]
dimention x=0.0,y=0.0,z=0.0
transformation x=0.0,y=0.0,z=0.0
material diffuse r=0.0,g=0.0,b=0.0
material ambient r=0.0,g=0.0,b=0.0
material specular r=0.0,g=0.0,b=0.0
shininess=0.0
```

Dimention signifies the height, length and width of the accessory in mm. Transformation defines where in the coordinate system the accessory is placed. The three material settings are for coloring and material shaping. shininess is also a cosmetic setting.

3.3 Functionality

The mpnguide application is seen on figure 7

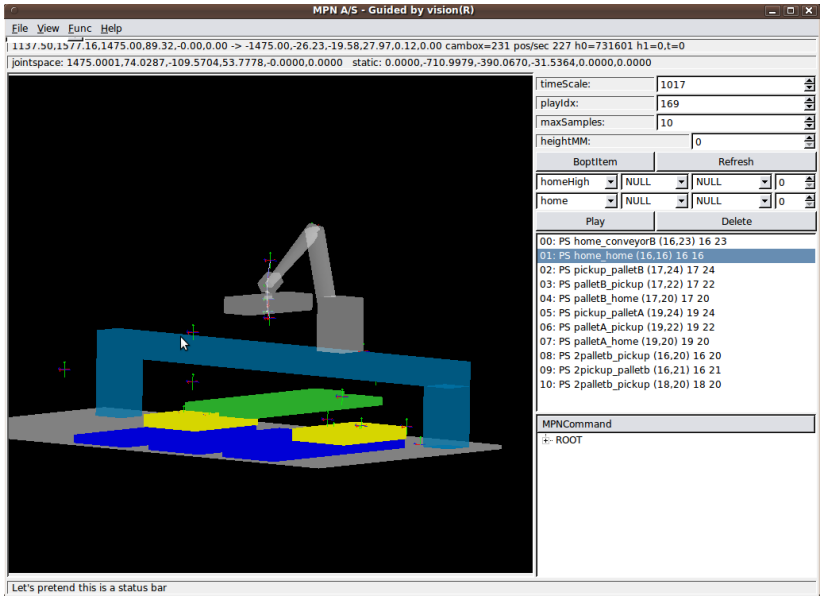


Figure 7: mpnguide software

Parameters of simulating a path:

Value	Functionality
timeScale	Gives an approximal time in ms (not exact)
playIdx	Shows which index number the path is in. This can be adjusted up or down to move the path forward or backward
maxSamples	number of samples per index
heightMM	Not working

Table 5: Functionalities of buttons

The buttons:

Button	Functionality
BoptItem	Lets the user choose a new path to simulate from a dropdown menu, afterward the refresh button must be pressed.
Refresh	Refreshes the boptcoeff file but not the frames
Play	Plays the path chosen in the table
Delete	Not working

Table 6: Functionalities of buttons

The four dropdown menu's between the buttons are for:

[From Frame ∇]	[From Item ∇]	[From Pattern ∇]	[Number of Items ∇]
[To Frame ∇]	[To Item ∇]	[To Pattern ∇]	[Number of Items ∇]

Table 7: Caption

3.4 Optimizing

When optimizing the paths in the boptcoeff file, the user has to apply the changes, save the file and then press the refresh button in order to see the new trajectory of the path. If the user needs to change the position of a frame, the mpnguide must restart in order to see it.

4 Robot Path execution

To load and execute paths, it's recommended to use one statemachine for controlling which paths the robot should use. As a standard the statemachine called RobotProgram.statemachine is the controlling statemachine.
The preload is essential for the program to be able to reset after shutdown or emergency stop. It's controlled in the ST_RESET of the RobotProgram, it should be linked from the loader.statemachine:

```
linkValue=resetpreload Loader
* Syntax *
linkValue=[ValueName] [Statemachine]
```

resetpreload's protocole is in Table 8.

Value	Action
0	The STM can signal a reset of preloaded path's by setting resetpreload to 1
1	The EXECUTER repsonds by setting resetpreload to 2
2	The LOADER responds by resetting preloaded path's and then setting resetpreload to 3
3	STM acknowledge by setting resetpreload to 0

Table 8: Resetting the preloader

Resetting the preloader in RobotProgram state ST_RESET:

```
State=ST_RESET
    TEST resetpreload = 0
       SET resetpreload 1
    ENDTEST
    TEST resetpreload = 3
       SET resetpreload 0
      SETSTATE ST_START
    ENDTEST
END
```

RobotProgram needs some standard values from Frames.statemachine:

```
linkValue=f_homeHigh  homeHigh   Frames
linkValue=f_home      home       Frames
linkValue=f_conveyor  conveyor   Frames
* Syntax *
linkValue=[LocalName] [ValueName] [Statemachine]
```

It must also have paths, **home_home** is the name of the path, the first number **16** is the mode, second number **20** is the pathtype:

```
Path=home_home        16 20
Path=home_conveyor    16 23
* Syntax *
Path=[PathName] (mode) (pathtype)
```

First the program has to load a path like in section 4.1. Then it must test if the path is loaded before executing in section 4.2. When the path is executing the statemachine must wait for the trio to finish in section 4.3.

A Path has states just like a statemachine, see Table 9.

State Name	State Number	State Type	Exit Condition
ST_IDLE	1	Inactive	LOAD
ST_LOADED	2	Inactive	EXEC
ST_FINISHED	3	Inactive	EXEC or LOAD
ST_ERROR	4	Inactive	N/A
ST_INACTIVE	5	Command	N/A
ST_LOAD	6	Command	LOADER loads path
ST_EXEC	7	Command	EXECUTER executes path
ST_BOPTING	8	Active	Change to ST_LOADING
ST_LOADING	9	Active	Change to ST_LOADED
ST_EXECUTING	10	Active	Change to ST_FINISHED

Table 9: Path states

4.1 Loading

The initial load command of the homeHigh to home path must be in a state for itself. To load a path, the command LOAD is used with this syntax:

```
LOAD home_home f_homeHigh f_home
* Syntax *
LOAD [Path] (startFrame) (endFrame)
```

4.2 Executing

First test if the wanted path is loaded. Before executing the loaded path, consider loading the next path like so:

```
TEST home_home.state = ST_LOADED
    LOAD home_conveyor f_home f_conveyor
    PRINT home_home
    EXEC home_home
    SETSTATE ST_AT_HOME
ENDTEST
* Syntax *
EXEC [Path]
```

For good debugging practice it's recommended to "print" the path before executing. Loading the next path will make the transition from each path smoother.

4.3 Finishing

To make sure that the path is finished so that the program can continue, the path state must be tested for ST_FINISHED:

```
TEST home_home.state = ST_FINISHED
    SETSTATE ST_MOVE_CONVEYOR
ENDTEST
```

4.4 Standard path handling

It's recommended to name the loading executing and finishing states, in the RobotProgram.statemachine, as corresponding to which path is involved. E.g.:

```
State=ST_LOAD_HOME
    LOAD home_home f_homeHigh f_home
    SETSTATE ST_MOVE_HOME
END

State=ST_MOVE_HOME
```

```
    TEST home_home.state = ST_LOADED
        LOAD home_conveyor f_home f_conveyor
        PRINT home_home
        EXEC home_home
        SETSTATE ST_AT_HOME
    ENDTEST
END

State=ST_AT_HOME
    TEST home_home.state = ST_FINISHED
        SETSTATE ST_MOVE_CONVEYOR
    ENDTEST
END
```