Final Report

25th of March 2015

2IO70

Version 1

This document will contain the documents of the preceding phases and give an introduction and conclusion to the project. "The Final Report presents the reader with a clear picture of the designed machine, the method of working followed, the specification, validation, and design of the software, and a motivation of the main design decisions." (Source: *Project Guide Design Based Learning "DBL 2IO70" "Sort It Out"*)

Group 16

Rolf Verschuuren

Wigger Boelens

Stefan van den Berg

Dat Phung

Maarten Keet

Tudor Petrescu

Introduction

In this document you will find the details on how we have designed and built, in the past eight weeks, a sorting machine and the software that runs it. This Final Report will contain the five “Product” documents previously handed in and approved by the tutor, and a “Process” document. The five Product documents explain how we arrived at our final design for both the hardware and the software. In order of when we made them, these are “Machine Design”, where we explain how the machine was designed and why we chose to do it that way. Subsequently comes “Software Specification”, where we made a finite state automaton that the software was going to be based on. Then came the “Software Design” and “Software Implementation and Integration” documents in which we first designed the full program in pseudo-Java code and then subsequently translated this into working Assembly code. Throughout this document there are validation segments in which we explain how we validated our decisions. In the “Validation and Testing” document we look back at these segments and describe the measures we took to ensure that our final product would meet the initial requirements. The second part of the Final Report is the Process document, in this document we describe how we worked as a group over the course of this project, and how we decided to tackle any issues that arose. This Final Report is the final deliverable for the course.

Table of Contents

Product

Machine Design

In this phase, we explain the design of our machine and how we decided on this design, and why we decided on this design. To do this we will take a look at our requirements and priorities. Afterwards we will look at the design and the decisions leading to that design.

High level Specification

The specification as given in the Technical Guide

The goal of this project is to build a simple sorting machine that is able to separate small objects, plastic discs that may be either black or white, into two sets: the black discs and the white discs. (...) The machine must contain at least one conveyor belt.
(...)
The machine is to be operated by means of two push buttons, called "START/STOP" and "ABORT" (...) By pressing button "START/STOP" the machine is started. (...) If 4 seconds after (...) expected arrival time the presence detector has not signalled the arrival of a disc (...) the machine stops (...) If, during the sorting process. The push button "START/STOP" is pressed the machine (...) continues its normal operation until the current disc has been deposited into the correct tray. Then, the machine stops. (...) Push button "ABORT" (...) makes the machine halt immediately. (...) Pressing this button while the machine is in its resting state has no effect. (...) If subsequently, the push button "START/STOP" is pressed once, the machine returns to its resting state.
To be able to guarantee that the mechanism depositing discs onto the conveyor belt stops in a well-defined state, this mechanism must be equipped with (at least) one switch to signal that this mechanism has reached the correct state.

Our specification

We have to make a so-called sorting machine. This machine should be able to separate, by colour, small black and white plastic discs. The requirements are as follows, the machine should:

- Have at least one conveyor belt.
- Have two buttons called "START/STOP" and "ABORT".
- Start when the machine is in a resting state and "START/STOP" is pressed.
- Stop when the machine is running and "START/STOP" is pressed, before stopping it should sort all discs that are on the belt.
- Abort when "ABORT" is pressed, this should halt the machine immediately unless it's in the resting state.
- Go to a resting state when the machine is in a halting state and "START/STOP" is pressed.
- Have at least one switch to signal when the machine is in a resting state.

Priorities

1. We define reliability as the ability of the machine to correctly sort all the inputted disks. We validate the reliability of the machine by checking the correctness of the code running the machine and also by conducting long-term test. Reliability is mainly reflected in our decision to encase the conveyer belt so that it is prevented any possibility of the discs, that are transported, to slip out. The goal of the project cannot be met with an unreliable design.

2. The speed of the machine is defined by the number of disks sorted in a unit of time. We search to select the design solution that improves this number. Speed is essential to offer a pleasant experience operating the machine. Speed is also the first thing that stands out when two machines of this sort are compared.

3. We define robustness as the fact that the machine does not break easily. The validation is if the machines state wouldn't be changed, they wouldn't break during: build phase, test phases, simulations, transportation and the end process, all during the period of the project cycle. Then we can consider the machine to be robust. Robustness can be observed from our design solution from the partial encasing used. Also the disc container was design to be robust do to its shape, size and simplicity. We do not meet our project goal if the machine isn't capable of running during the final process.

4. We define user accessibility as the ease in which the user takes the actions required from the machine. Validation is done by checking the compatibility of the design and the user constrains. The disc container was built with user accessibility in mind, it is fairly easy and fast to load discs. The reason why this priority is important is that the machine requires a user to be operated and in consequence its operation must be possible.

5. We define amount of space by the amount of floor space that the machine occupies. Checking if there are useless components in the machine or other components that can be replaced with smaller counterparts without influencing the priorities above does validation of the low amount of space. From this perspective the current Feeder occupies a small amount a space, while the other feeder design would of forced us to add an extra floor extension because of its large dimensions. The reason of this priority is to ease the transportation and storage of the machine.

6. The Difficulty of Building is self-explanatory. We validate this be checking if there are any useless components. In our decision to have the conveyer belt larger, trying to fit on the platform size, we simplified the design and left more physical space to work on the other components connected to the machine. Opting for such a priority would make our solution easy to implement.

7. The Amount of Parts of the Machine is also self-explanatory. We also check if there are any useless parts. An example were we used very little parts by choice in our machine is the feeder component. Reasons why we picked this priority is that it might improve the overview of the machine and also the error-detection.

For the validation of these priorities see "Testing machine design to the priorities".

System Level requirements

The system level requirements consist of 3 parts. These 3 parts are the USE-cases, the safety properties and the user constraints.

USE-cases

There are 6 USE-cases, which are described below.

Starting the machine

Primary Actor	Machine operator (student or teacher at Tu/e)
Scope	A sorting machine
Brief	The machine operator starts the machine, machine parts go to their initials state and the machine starts sorting.
Postconditions	The machine starts the sorting process.
Preconditions	-
Trigger	Booting the machine / finished the abort or start/stop routine
Basic Flow:	1. Machine puts devices in their initial state. 2. The user presses the START/STOP button

Stop the machine

Primary Actor	Machine operator (student or teacher at Tu/e)
Scope	A sorting machine
Brief	The machine is waiting for the current process to end before it is send into an inactive state.
Postconditions	The machine is sent into an inactive state with no process interrupted.
Preconditions	The machine is running.
Trigger	The START/STOP button is pressed.
Basic Flow:	1. The machine finishes sorting the disks currently in the machine 2. The machine enters an inactive state and will not take any more disks form the storage* unless the START/STOP button is pressed

Sort unsorted disks

Primary Actor	Machine operator (student or teacher at Tu/e)
Scope	A sorting machine
Brief	The machine sorts the unsorted disks provided into two separate containers based on colour.
Postconditions	There are no unsorted disks left All sorted disks are in a container based on their colour
Preconditions	The machine is not already running.
Trigger	The user provides unsorted disks and presses the “START” button.
Basic Flow:	1. An unsorted disk is moved to the colour detector 2. The machine decides to which of the two containers the disk needs to be moved 3. The machine moves the disk to the designated container 4. The machine repeats step 2 through 4 until all disks have been sorted 5. The machine pauses within 4 seconds

Abort the process

Primary Actor	Machine operator (student or teacher at Tu/e)
Scope	A sorting machine
Brief	The machine should immediately stop doing anything.
Postconditions	The machine stopped running and is ready to start again.
Preconditions	The machine is sorting discs.
Trigger	The use wants to immediately stop the machine.
Basic Flow:	1. The machine stops transporting the discs. And doesn’t put any more discs on the transporting mechanism. 2. The user is required to remove all discs that are neither in the container unit nor sorted. 3. When the user removed all unsorted discs that were not in the container unit he presses the START/STOP button.

Booting of the machine

Primary Actor	Machine operator (student or teacher at Tu/e)
Scope	A sorting machine
Brief	The machine will prepare to start the program. And do the required actions.
Postconditions	The machine is ready to get instructions of the user.
Preconditions	The machine is off.
Trigger	N/a
Basic Flow:	1. Connect the PP2-board to the pc. 2. Plug the pp2-board in to the power socket. 3. Start the debugger 4. Connect the pp2-board using the debugger. 5. Load the program into the debugger. 6. Run the program.

Shutting down the machine

Primary Actor	Machine operator (student or teacher at Tu/e)
Scope	A sorting machine
Brief	User unplugs the power supply and disconnects the processor from the PC and the machine.
Post conditions	The PC can be used for other things and the processor and machine can be stored separately.
Preconditions	Everything is in its initial state or the machine has stopped.
Trigger	N/a
Basic Flow:	1. Unplug the power supply of the machine. 2. Unplug the power supply of the processor. 3. Disconnect the processor from the machine. 4. Disconnect the PC from the processor.

User Constraints

- Before the start button is pressed, the user is required to place all discs to be sorted in the container unit
- While the machine is running the user is not allowed to move the machine or touch anything except the buttons.
- When the abort button is pressed or the machine has been shut down, the user is required to remove all discs that are neither in the container unit nor sorted.

Safety Properties

1. After pressing an emergency button, within 50ms there should be no moving part in the machine
2. If all disks are sorted the machine should stop within 4 seconds.
3. After the start-up of the machine, the assembly program should not stop until the machine is shut down.
4. The outputs connected to the h-bridge may never be powered on at the same time.
5. The outputs connected to the motors should never output more than 9 volts

Explanation of Safety Properties

1. When there is an emergency it is important that whatever is going wrong will not get worse. One of the ways this can happen is for instance that someone's finger gets stuck, to minimize damage to this finger the machine should stop quite fast. After discussion we decided 50ms would be a reasonable maximum stop time as it whatever is going wrong will not get worse in 50ms.
2. To minimize electricity usage we think that the machine should not keep running while there are no disks in it.
3. If the assembly program stops while the machine is still running, we can no longer control the machine. We can for instance no longer detect when the emergency button is pressed, meaning we cannot guarantee safety property #1.
4. The H-bridge should never have two inputs powered on at the same time. Because then you create a short circuit.
5. According to the project guide this is the maximum voltage the motors are certified to work with.

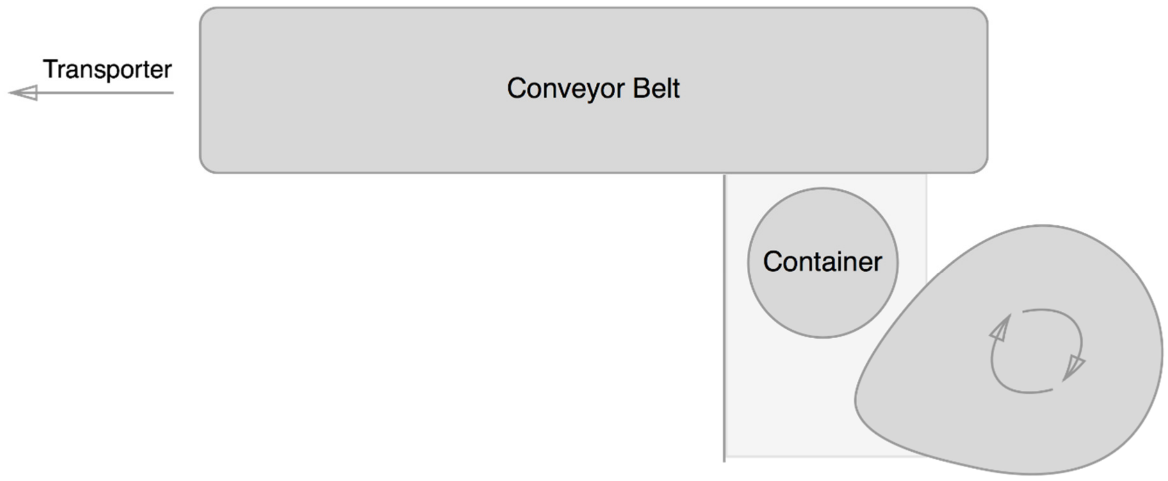
Design Decisions

The way we approached the design of the machine is by separating the machine into multiple parts. Those parts exist out of: the feeder, the transportation mechanism, and the sorter.

The Feeder

The feeder has as objective that it needs to somehow get the disks from the container onto the conveyor belt. This is needed for the use case “Sort unsorted disks”.

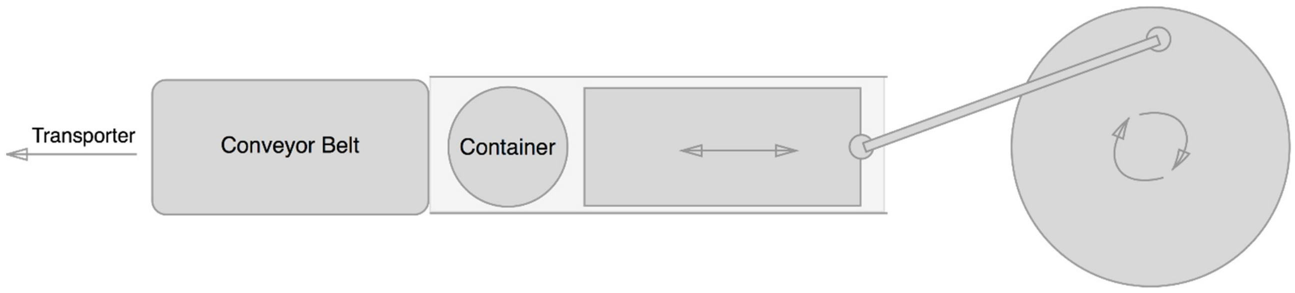
For the design of this feeder we had two competing designs. Both use the two hollow tubes stacked as a container. We chose to do this because they are completely reliable in containing the disks and because a new disk simply falls out if the bottom one is removed, they are very fast. Because the container is made off two big parts and some small parts to make them stack, the container is also very robust. It’s quite easy to put the disks into the big hole at the top, so user accessibility was very high. In short, the first solution that came to mind scored extremely high on all priorities and we looked no further.



The first design for the feeder consist of 3 important parts. First you have the container. The container drops a disk, which is then pushed onto the conveyor belt using a cam. A wall to the left of the container makes sure the disk is pushed up and not to the left.

Our second feeder design also consisted of a block that pushes the disk. To make this block move a lever attached to a wheel is used. Rotating the wheel makes the block move back and forth, pushing disks onto the conveyor belt.

Both designs correctly implemented the use cases. To test which one would be better we



build both and tested them. They scored the same on almost all top priorities. They were both completely reliable for instance. There was also no difference in speed, both would push a disk onto the conveyor belt with every turn of their wheels. Both did not hinder the user, so the good user accessibility of the container was unchanged. When we came to the last three priorities there were some differences making us choose the first design: It was easier to build, used less parts and was a lot more compact.

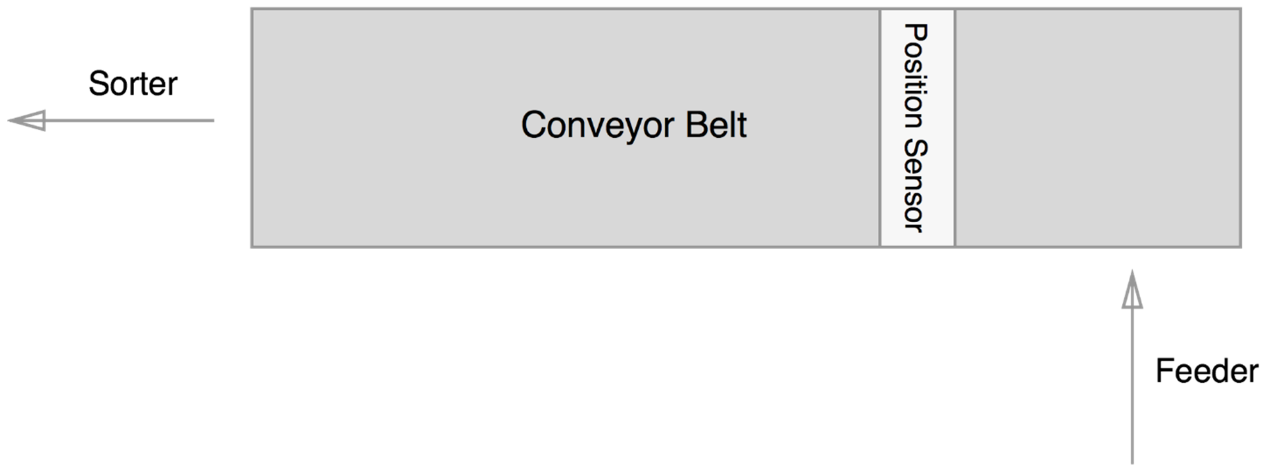
The Transportation and Scanning

When considering the transportation method we had a 3 main ideas. The first one was that we used a short conveyor belt. The second idea was about a long conveyor belt. And the last idea used a turning wheel and 2 conveyor belts. All these ideas included a conveyor belt because that was required.

The thought behind the short conveyor belt was that in the feeding mechanism would push the discs hard enough so that we could put the sensors on that part and to have a small but conveyor belt to transport the discs. The conveyor was short because nothing needed to happen on it. Thus it would only be there because it was a requirement. To us it seemed a bit useless to not do anything on the conveyors belts. So that was when the second arose.

The second idea had a long conveyor belt to put the sensors on. And also a part of the separating mechanism. The conveyor belt would limit how fast the machine can run but all the actions would happen on the conveyor belt so that time wouldn’t be wasted. It also isn’t that hard to create a long conveyor belt so we kept the idea in mind.

Our final idea was that there would be some sort of wheel with separate compartments for discs in the centre which would rotate and put discs on to two different conveyor belts. Each conveyor belt led to a storage unit of the sorted discs. The problem with this idea was that it would be hard to prevent the discs from spinning out of the compartments when they shouldn't while still being able to let the discs go out when they had to. Because we couldn't get it to work the idea was dropped and we went back to the idea about a long conveyor belt.



We were capable of realizing the of the long conveyor belt. But during the build of the conveyor belt we noticed that it would not be tight enough around the gears. Thus we tried to remove a small part of the belt. But this still didn't have to effect we hoped for. So we added a third gear in the middle which tightened the belt to an acceptable state.

The conveyor belt was still far from perfect because it would tilt at certain points and the discs could fall off. So to prevent it we build 2 walls around the belt. On the first part they are low because the low walls were more robust than the high walls and for the user it is easier to access the discs on the conveyor belt. The high walls have been secured using 4 pillars because that made it robust enough to make sure they didn't break. The walls had to be high because we needed to put a set of sensors on it.

Those sensor had to be above the conveyor belt. They also needed to be at an angle to work properly. That was required else the sensor wouldn't be able to check if the disc was black or white.

The other set of sensors didn't need to be place at an angle thus they were simply put on each side of the conveyor belt. This set of sensor would then be capable to scan if there was a disc on that spot of the conveyor belt. This sensor is need to time at which moment the other set of sensor had to check the colour of the disc. And it is also used to check if there are any more discs left to scan.

The sorting mechanism

For the mechanism that does the actual sorting we chose between a couple of different designs. These designs are listed and explained below.

The first, and most simple design was to use just one conveyor belt that would move left or right based on the colour of the disks. This design is listed under the use of the conveyor belt above, this is why I will not describe it again.

The second design is a slight improvement on the first one where we would use a second, shorter, conveyor belt to do the sorting. This design would place the two conveyor belts in a T-shape with the colour check done on the first one, after which the second conveyor belt moves left or right. We considered this design an improvement on the first one because the second conveyor belt could be made much shorter. This means that the design can sort faster than the single conveyor belt one.

The second conveyor belt was faster than the first design with only one belt, however we soon realized that we could do this even faster. By removing the second belt and replacing it with a seesaw that could be angled to face one of the two sorted containers, we could increase the speed even more. Since the disk would essentially be sorted the moment it reached the end of the conveyor belt. This would be a great design, was it not for the fact that the seesaw required a lot of height. In fact, the entire machine looked like it was placed on stilts, requiring us to use lots of parts and having a lot of wasted space underneath. This design could do it faster at the cost of requiring more space than any of the others.

While the use of a seesaw sped up the sorting process, it also took a lot more space, so we went back to the drawing board and discarded this idea. Instead coming up with a wedge that would be slide onto the conveyor belt from the side whenever a disk of a certain colour is detected. This would then allow the conveyor belt to push the disk against the wedge making a roughly 45° angle thus pushing the disk of the side of the belt and into the collection box. The second colour could just continue while the wedge was pulled back and off the end of the belt. This means that the design cuts off part of the machine at the end and allowing us to make the machine lower than before.

We liked the idea of letting the conveyor belt doing the sorting by placing a wedge in the way, but after some thinking we realized that it could be done both faster and more compact. The trick was to change the direction in the wedge moves from horizontal to vertical. Doing so moves the entire mechanism, aside from the wedge itself, in an upright position pushing it very close to the machine. Aside from saving space, this also allowed the wedge to move much less, since it only has to move just over 1cm above the conveyor belt rather than move all the way over it to the side. This final design does not sacrifice any reliability from its predecessors while being the fastest. It also takes by far the lowest amount of floor space, characterized by the fact that this final design including this sorting mechanism is our only design that fits on only one of the two provided floor plates. For these reasons we believe this design for the sorting mechanism to be the best.

Machine interface

The feeder
The motor for the feeder turns a clam. With that motor turning clockwise the disc, which is on the surface in front of the clam, will be pushed off the surface and on to the conveyor belt. To make sure the engine runs clockwise the minus has to be connected to the connection closest to the spot where 6V is marked. We connect this engine to the 3rd output of the pp2-processor.

The position sensor
The way a position sensor is set up us by using a lens lamp and a phototransistor. The lens lamp will be shining in the direction of the phototransistor. The light from the lens lamp makes the phototransistor send a signal to the pp2-processor. If a disc comes in between the lens lamp and the phototransistor then there won't shine any light at the phototransistor and thus it won't send a signal to the pp2-processor. The phototransistor is connected to the 8th input of the pp2-board. The phototransistor is polarized and thus it is important that it is connected correctly. The correct way to connect is with the ground to the connection closest to the white spot on the phototransistor. The lens lamp isn't polarized and does not move in any direction and thus it doesn't matter in which connection the ground is. The lens lamp is connected to the 2nd output of the pp2-processor.

The black white detector
The black white detector uses the same components as the position sensor but they are implemented in a different way. The way in which the colour is detected is by the reflection of light on the disc. Because white discs reflect light very well the phototransistor does pick up some light and thus sends a signal. Black disc on the other hand do not reflect enough light to let the phototransistor pick it up. Thus a white disc can be detected if the sensors are placed in the correct way.

To make sure the phototransistor picks up only the reflected light a cap is placed over it with a hole in the middle. So only light from in front of it will influence the phototransistor. But to make sure that the reflected light can pass through that hole the sensor must be placed at an angle. The reflected light, which is detected by the phototransistor, is at its strongest when the lens lamp is also placed at an angle.

We connected the lens lamp in the same way as the lens lamp of the position sensor only now to the 6th output of the pp2-processor. The phototransistor is also connected as described in the position sensor only now to the 3rd inputs.

The Sorter

The divider uses a so-called “H-bridge” to move up and down. We use output 0 and output to control the H-bridge, which in turn controls the motor moving the divider. We connect the ground of the H bridge with the output 0 to the 6-side of the motor. Now when we power up output 0 the divider will move up. When we power up output 1 the divider will move down. Output 0 and output 1 are never allowed to be on at the same time, which is also stated in the safety properties. We want to move the divider as fast as possible so we always use the maximum allowed voltage of 9 volts. To detect when the divider is in its upmost position we use a push sensor. When the PP2 detects that this push sensor is pressed we immediately cut the power to output 0. We do not detect when the divider is at the bottom, because as soon as the push sensor is not pressed then there isn’t enough space for a disc to go underneath. Thus we simply power on the motor for a set amount of time. This time should be enough to make it move to the bottom but not low enough to interfere with the conveyor belt.

The buttons

The button that is used to start/stop the machine will be button 0. The button to abort the machine will be button 1.

The conveyer belt

The conveyer belt uses 5 gears of which only 3 touch the conveyer belt. 2 of those 3 gears are used to make sure the conveyer belt is horizontal and the third one is used to make the conveyer belt turn. The third gear is connected to a metal rod. On that metal rod another gear is connected and that gear will be turned using the gear which is connected to the engine. Because we have those gears in between the direction in which the engine turns has to be counter clockwise. Then the conveyer belt does turn clockwise and the discs will be moved in the right direction. To let the engine turn clockwise we have to connect the ground to the connection closest to the 9V. This engine is connected to the 3rd output.

System Validation and Testing

Validate High level specifications

Our high level specifications are correct, because in the exercise it is said that a sorting machine for black and white discs should be made. And it also is said that we need at least one conveyer belt.

Validation SLR

The high level specification defines the basic flow of the use-cases, user constraints and safety properties. At the same time, we validate the System Level Requirements through the high level specification. “Sort unsorted discs” is correct, because the high level specification mentions that the machine should sort discs. Aborting the process happens because in every machine something could go wrong and thus it needs to be able to be stopped at any point in time. “Starting the machine” and “Stopping the machine” are actions which are also needed for machines because else you couldn’t make them stop or start doing what they are supposed to do. “Booting up the machine” and “shutting down the machine” is required, because the disc sorter has to be turned on and off, in order for it to fulfil its purpose.

Before the start buttons is pressed the user is required to place all discs to be sorted in the container unit. The discs should be placed in the container, so that the machine is able to sort the discs.
While the machine is running the user is not allowed to move the machine or touch anything except the buttons. If the user makes contact with either the conveyor belt or the discs while they’re on the conveyor belt, the machine might not be able to separate the discs correctly.
When the abort button is pressed or the machine has to be shut down, the user is required to remove all discs that are neither in the container unit nor sorted. The user is supposed to do this, so that the machine will be able to restart the sorting process with a new disc.

After pressing an emergency button, within 50 ms there should be no moving parts in the machine. The machine should immediately abort its current process, according to the high level specification, although this is not realisable. Therefore, this is set to be within 50 ms.

According to the High level Specification the machine should stop sorting if there is no more disk signalled after 4s. We made this into a safety property, because a running machine with no use is only going to possibly harm people getting in contact or the machine itself.

According to what the high level specification offer, there is nothing that could stop the assembly program as long as the code is correctly written for this purpose, we don’t consider accidents and flaws, the only way for the program to end is by powering off the machine.

The outputs connected to the h-bridge may never be powered on at the same time. If this happens, the PP2 processor short circuit, and the machine won’t work anymore.

Validation Priorities to SLRs

Reliability:

The use-cases describe how we want to sort multiple coloured disks, because we want the sorting to be done as accurately as possible we chose reliability as one of our priorities.

Accessibility:

The use-cases describe that the user has to remove all disks from the machine after the "ABORT" button is pressed. Because of this we want to make the machine somewhat open, so the user can remove the disks with relative ease.

Speed:

The use-cases describe how we want to sort multiple coloured disks, because we want the sorting to be done as fast as possible we chose speed as one of our priorities.

Robustness:

The use-cases describe that the user has to remove all disks from the machine after the "ABORT" button is pressed. For this reason we want the machine to be fairly durable so that the user does not easily damage it. Additionally, since the machine contains a number of engines and moving parts, it will be vibrating ever so slightly. These vibrations should also not cause any damage to the machine leading to our priority of robustness.

Amount of space:

This priority does not have a clear relation to our SLRs, however, we believe that a small machine capable of accomplishing the same task is generally better than a larger version. This is because the machine has to be stored or placed somewhere, leaving you with more space for other machines. This is why we chose for minimizing floor space as one of our priorities.

Difficulty of building:

This priority also does not have a clear relation to our SLRs, but this would make our job as builders easier. It would also allow for greater rates of production of the machine. For these reasons we chose difficulty of building as one of our priorities.

Amount of parts:

This priority also does not have a clear relation to our SLRs. A lot of parts, though, would make our machine more expensive and harsher on the environment, leading us to make the amount of parts one of our priorities.

Because the priorities "Amount of space", "Difficulty of building" and "Amount of parts" have no clear relationship to the SLRs we chose to put them on the bottom of our priority list.

Testing machine design to the priorities

1. Perform a test with alternating black and white discs to test the moving of the divider multiple times and check that the discs are sorted right and all discs were sorted.

2. Check if it sorts 10 discs within 30s with a load of white discs, black discs and alternating black and white discs

3. Let the machine perform a run without pushing buttons and with pushing the abort button while running and check if nothing breaks.

4. Look at points in the machine where a disc could get stuck and check if you can access the disc to remove it.

5. Check if the machine fits on 1 floorboard of the Fischer Technik.

6. Check if you can build the machine within 1.5 hours with 2 people.

7. Check if there are any parts without a function.

Software Specification

In the Software Specification phase, we give an as accurately as possible description of the required behaviour of the PP2, without describing how this is achieved, and a UPPAAL model of this behaviour. In order to do this, we translate the system level requirements to a high level specification of what the software controlling the physical machine should do.

Inputs and Outputs

Inputs

Input	The range/type of the value
Start/Stop button	Boolean value
Abort button	Boolean value
Push button(sensor)	Boolean value
Position sensor	Boolean value
Colour detector	Boolean value
Timer	Integer, values range from seconds to clock ticks, consists of TEnd, Motor Down, Motor Up, Sort, Belt, and Tic

The Start/Stop and Abort buttons speak for themselves. They are either pressed or not pressed.

Push button(sensor): the sorter touches the push sensor or doesn't touch it, to detect the sorter's position.

The position sensor and colour detector are either on or off.

Timer

The timer is a count-down timer that is set to a certain value and runs at a frequency of 10 kHz. All given times were calculated by taking the average time of ten measurements, using 50 to 60% of the Potentiometer on the PP2 board. Thus, the sorting mechanisms are faster in reality. The input of a timer is set to a defined value or not set.

TEnd is the moment of termination of the timer, so when the timer reaches zero.

Motor Down is defined as the time it takes for the engine of the sorter to move the sorter from the lowest point to the highest point, until sorting mechanism touches the push sensor. This takes 0.30 seconds.

Motor Up is the state of the sorter moving from the highest point to the bottom of the engine sorter. Since the engine sorter for Motor Down and Motor Up have the same voltage, this will take 0.30 seconds as well.

Sort is the amount of time it takes for a disc to be transported from the black/white detector to the end of the conveyor belt, which is measured to be 0.85 seconds.

Belt is the period that a disc travels from the feeder to the end of the conveyor belt, until the disc reaches the tray for black discs. This action takes 2.0 seconds.

Tic is defined as one clock tick of the PP2. A clock tick is incredibly fast.

Outputs

Output	The range/type of the value
Lens lamp position	Boolean value
Lens lamp sorter	Boolean value
Conveyor engine	Between 6 and 9 V while running 0 V when not running
Feeder engine	Between 3 and 7 V while running 0 V when not running
Hbridge0	Boolean value, current is between 6 and 9 V while running
Hbridge1	Boolean value, current is between 6 and 9 V while running
Display	Integer value, positive
Timer start	Integer, Values range from seconds to clock ticks, consists of TEnd, Motor Down, Motor Up, Sort, Belt, and Tic

Lens lamp position and **lens lamp sorter** are the lamps that make up part of the sensors and can be turned on or off.

The **conveyor and feeder engines** respectively move the conveyor belt and the feeder. They are either on or off.

Hbridge0 indicates whether the sorter moves up or not. On the other hand, whereas **Hbridge1** shows that the sorter moves down or halts.

The **display** shows the state that the machine is currently in. Depending on the available time, we might or might not implement this.

The **Timer start** output is the same as the Timer input, except that the timer counts down.

Validation of "Inputs and Outputs"

We see that the inputs and outputs of Software Specification are correct. The inputs of Machine Design should be equal to the outputs of Software Specification, which they are.

Relations

Lens lamp of the black white detector

The lens lamp of the black white detector will be on when the machine is sorting. Thus the lens lamp will react to the input of the "START/STOP" button and the "ABORT" button. The lens lamp will go on when the machine is in resting state and the "START/STOP" button is pressed and it will go off when the "ABORT" button is pressed while the machine was running.

Lens lamp of the position sensor

The lens lamp of the position sensor reacts only to the "START/STOP" button and the "ABORT" button. The lens lamp will be on after the "START/STOP" button is pressed and the machine is in its resting state. If at any other point in time the "ABORT" button is pressed it will go off. When the "START/STOP" button is pressed and the machine is running then the lens lamp also goes off.

Engine of the conveyor belt

The engine of on the conveyor belt only reacts to the input of the "START/STOP" button and the "ABORT" button. The engine will start when the machine is in its resting state and the "START/STOP" button is pressed. If however the "START/STOP" button is pressed and the machine is not in its resting state then the machine will stop after it completed its current cycle. Whenever the "ABORT" button is pressed the engine stops within 50ms.

Engine of the feeder

The engine for the feeder also only reacts to the input of the "START/STOP" button and the "ABORT" button. This engine also starts when the machine is in its resting state and the "START/STOP" button is pressed. If however the machine is running then the engine will stop. When the "ABORT" button is pressed the engine stops within 50ms.

Engine for the sorter

When the machine is running the engine of the sorter reacts to inputs of the colour detector, the push sensor and the timer. When a signal is received from the colour detector the engine pushes the sorter up, the engine then waits until the timer gives a signal to go down again after it let the discs through, it knows when it is in the correct "up" position from the push sensor . If the "START/STOP" button is pressed when the machine is in its resting state, then the sorter will wait for a signal from the timer that marks the end of the current cycle. If at any time the ""ABORT"" button is pressed, the sorting mechanism is to stop within 50ms.

Display for counting

The display output depends on how many times the colour detector detects a white disc and how many times a disc passes the position sensor without the colour detector detecting it.

In the initial state the counters get reset.

Validation of “Relations”

The relations between the inputs and outputs can be validated with the input/output tables. For all inputs, we have outputs. These outputs depend on one or more inputs, which is described in the Relations.

Design Decisions

Feeder
The feeder in constantly on because of priority 2, speed, mentioned in the Machine Design document. Another reason is that there's a turning part that needs to spin through to get to its initial position to be able to deposit discs again.

Lens lamp position
We chose to have the lens lamp for position sensor constantly on, because it's easier to code resulting in spending less time on it. The optimization is minimal if we would turn them off every time there's a gap between discs, because of the feeder being quite fast in depositing the next disc.

Conveyor belt
The conveyor belt is constantly running, because the feeder is constantly pushing discs onto the conveyor belt. This goes hand in hand with our second priority, which is speed.

Lens lamp colour
Like with the position sensor, it's easier to code that it is continuously on. The light being off if it's possible, would again be a minimal improvement, because the gaps between discs being pushed on the conveyor belt is the same as with the black white detector.

Push button
We use the push button, because of priority 1, correctness, to know if the sorter arm is at its highest point. We need to know this, because we need to know when to stop the motor making the sorter arm going up.

Description of States

Initial_state

In the initial state the machine starts calibrating the sorting mechanism by moving it up.

Outputs	Value for output
Lens lamp position	0
Lens lamp sorter	0
Engine conveyor	0
Engine feeder	0
Hbridge0	0
Hbridge1	0
Display	0
Timer start	0

Calibrate_Sorter

In the calibrate sorter state the sorting mechanism moves down until it is just above the conveyor belt.

Outputs	Value for output
Lens lamp position	0
Lens lamp sorter	0
Engine conveyor	0
Engine feeder	0
Hbridge0	0
Hbridge1	1
Display	0
Timer start	0

Resting_state

In the resting state the sorting machine is at rest and waiting for the user to press the START/STOP button.

Outputs	Value for output
Lens lamp	0
Lens lamp	0
Engine conveyor	0
Engine feeder	0
Hbridge0	0
Hbridge1	0
Display	0
Timer start	0

Running_state

In the running state the sorting mechanism, the conveyor belt, the position detector, and the colour detector are turned on.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	0
Hbridge1	0
Display	0
Timer start	2 s + Belt

Running_Wait

In this state a disc has been detected and that disc is moving along the conveyor belt to the sorter.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	0
Hbridge1	0
Display	0
Timer start	2 s + Belt

Running_Timer_Reset

In this state a new disc was detected and the timer has been reset.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	0
Hbridge1	0
Display	0
Timer start	2 s + Belt

Motor_Up

In this state the motor of the sorter is moving up until it hits the push button.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	1
Hbridge1	0
Display	0
Timer start	Sort

Motor_Up_Stop

In this state the motor of the sorter is moving up until it hits the push button. And the machine has to stop because the start stop button was pressed.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	1
Hbridge1	0
Display	0
Timer start	Sort

Motor_Down

In the Motor_Down state, the sorter is moved down.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	0
Hbridge1	1
Display	0
Timer start	0

Motor_Down_Stop

In Motor_Down_Stop, the sorter is moved down, after the start/stop button has been pressed.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	0
Hbridge1	1
Display	0
Timer start	0

White_Wait

In this state the machine waits until the colour detector has detected a white disc.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	0
Hbridge1	0
Display	0
Timer start	Sort

White_Wait_Stop

In this state the machine waits until the colour detector has detected a white disc, after the START/STOP button has been pressed.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	1
Hbridge0	0
Hbridge1	0
Display	0
Timer start	Sort

Running_Timer

Running_Timer is the state that sets the interrupt timer to make sure the machine stops after the current cycle.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	0
Hbridge0	0
Hbridge1	0
Display	0
Timer start	Belt

Motor_Up_Timer

Motor_Up_Timer is the state that sets the interrupt timer to make sure the machine stops after the current cycle.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	0
Hbridge0	0
Hbridge1	0
Display	0
Timer start	Belt

White_Wait_Timer

White_Wait_Timer is the state that sets the interrupt timer to make sure the machine stops after the current cycle.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	0
Hbridge0	0
Hbridge1	0
Display	0
Timer start	Belt

Motor_Down_Timer

Motor_Down_Timer is the state that sets the interrupt timer to make sure the machine stops after the current cycle.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	0
Hbridge0	0
Hbridge1	0
Display	0
Timer start	Belt

Aborted

Aborted is the state where the machines goes to if the abort button is pressed, the machine has come to a halt.

Outputs	Value for output
Lens lamp position	0
Lens lamp sorter	0
Engine conveyor	0
Engine feeder	0
Hbridge0	0
Hbridge1	0
Display	0
Timer start	0

Running_Stop

Running_Stop gives the same outputs as the Running state, the only difference being a running timer in the stop process.

Outputs	Value for output
Lens lamp position	1
Lens lamp sorter	1
Engine conveyor	1
Engine feeder	0
Hbridge0	0
Hbridge1	0
Engine sorter	0
Display	0
Timer start	Belt

Validation of “Description of States”

To validate the states we will look at the USE-cases again to see if every USE-case is implemented. To do this we look at the basic flow and trigger of every use case and see what states we use to realize this.

We also validate the states to the relations. For every USE-case we looked at what states would be necessary to achieve it.

Starting the machine

Preconditions: -
Trigger: Booting the machine / finished the abort or start/stop routine
Postconditions: The machine starts the sorting process.

Basic Flow	State	Explanation
Before Trigger	Any State	It does not really matter which state the machine is in before the trigger
After Trigger	Initial State	Initial state is the first state, so after booting the machine we will be here. Finishing the abort or start/stop routine will also end in the initial state
1. Machine puts devices in their initial state.	Initial State + Calibrate Sorter + Resting State	The only thing that needs to be put into an initial state is the sorter mechanism. In initial state the machine moves the sorter up until it touches the push button. It then transitions to Calibrate Sorter where it starts moving down. After a set amount of time it will stop moving the sorter and transition to the resting state. This way we know exactly where the sorter is positioned
1. The user presses the START/STOP button	Running State	From the Resting State the transition to the running state is pressing the START/STOP button
Postconditions	Running State	The running state is the start of the sorting process

Stopping the machine

Preconditions: The machine is running.
Trigger: The START/STOP button is pressed.
Postconditions: The machine is sent into an inactive state with no process interrupted.

Basic Flow	State	Explanation
Preconditions	Not initial state, Calibrate Sorter or aborted	When the machine is not in any of these states it is running.
After Trigger	One of the (greenblue) Timer states	When the START/STOP is pressed the machine transitions to a timer start state, which starts a timer and stops the feeder mechanism.
1. The machine finishes sorting the discs currently in the machine	One of the sorting states	While the timer is running the machine keeps sorting. The timer is the time it takes for the conveyor belt to make a complete rotation, guaranteeing there are no more discs on the belt.
1. The machine enters an inactive state and will not take any more discs form the storage* unless the START/STOP button is pressed.	Initial State + Calibrate Sorter + Resting State	After going through the initialize process we go back to the resting state, which waits on the START/STOP button.
Postconditions	Resting State	Resting state in an inactive state and we finished the sorting process.

Sort unsorted discs

Preconditions: The machine is not already running.
Trigger: The user provides unsorted discs and presses the "START" button.
Postconditions: There are no unsorted discs left, all sorted discs are in a container based on their colour.

Basic Flow	State	Explanation
Preconditions	Resting State	The program first initializes and then waits for the user to press that start button. This waiting happens in the Resting State. In the resting state the machine is not running
After Trigger	Running State	Pressing START/STOP is the input to transition to the running state
1. An unsorted disc is moved to the colour detector	Running Wait + Running Timer Rest	When moving to the colour detector it will have to pass the position Sensor which is the input to move to Running Wait, the disc is then still in front of the position sensor so the program moves to Running Timer Rest
1. The machine decides to which of the two containers the disc needs to be moved	Running Wait + Running Timer Rest OR Motor Up + White-Wait	Depending on whether the disc is white or black the sorter either needs to move down or keep its down position. If it keeps its down position it should just keep checking for an unsorted disc and when it detects one it will move to Running Timer Rest If it needs to move up the colour detector will detect a white disc and therefore transition to Motor Up. Moving the sorter up will trigger the pushButton, which is the input to transition to White-Wait
2. The machine moves the disc to the designated container	Running Wait + Running Timer Rest OR Motor Down + Running Wait	If the sorter did not detect a white disc we are still waiting like in basic flow 2. If it did detect one then while the disc is moving to the designated container the sorttimer will count down making the machine transition to Motor Down
3. The machine repeats step 2 through 4 until all discs have been sorted	-	
4. The machine pauses within 4 seconds	Initial State + Calibrate Sorter + Resting State	If there are no discs anymore the machine will stay in Running Wait waiting for the timer interrupt which will come within 4 seconds, making the machine transition to initial state. There it will reset the sorter and transition to the resting state
Postconditions	Resting State	We repeated the sorting step until all discs where sorted, meaning all discs are now sorted

Abort the process

Preconditions: The machine is sorting discs
Trigger: The user wants to immediately stop the machine.
Postconditions: The machine stopped running and is ready to start again.

Basic Flow	State	Explanation
Preconditions	Every that is not initial state, Calibrate Sorter, resting state or Aborted	All other states are states in which discs are being sorted
After Trigger	Aborted	Every state (apart from the one mentioned in before trigger) have a line to abort with Abort as input
1. The machine stops transporting the discs. And doesn't put any more discs on the transporting mechanism.	Aborted	Because the machine is now in the abort state, which has all outputs set to 0, nothing will be moving.
1. The user is required to remove all discs that are neither in the container unit nor sorted.	Aborted	The machine will remain in Abort until the user presses START/STOP. This means everything is stopped and the user can safely remove all discs
2. When the user removed all unsorted discs that were not in the container unit he presses the START/STOP button.	Initial State + Calibrate Sorter + Resting State	Pressing the START/STOP button is the input for the transition to Initial State There it will reset the sorter and transition to the resting state
Postconditions	Resting State	We are in the resting state, so the machine has stopped running. The resting State is also the state from which you can start the machine again

Booting of the machine and Shutting down the machine do nothing with our software. This means they do not use states. This also means we can't validate those USE-Cases here.

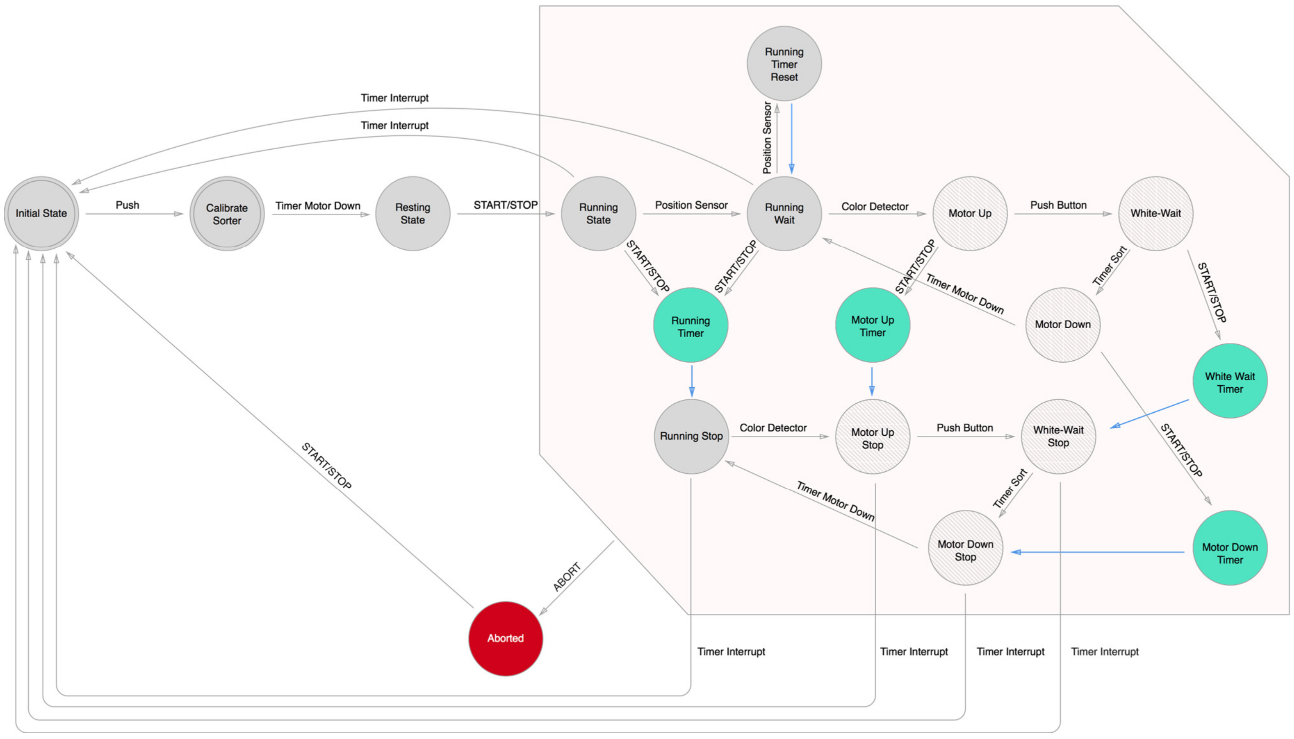
State transitions

Current state	Input	Input value	Next State
Initial	Push	1	Calibrate_Sorter
Calibrate_Sorter	Push	0	Resting
Resting	StartStop	1	Running
Running	Timer	TEnd	Initial
Running	PositionSensor	0	Running_Wait
Running	Abort	1	Aborted
Running	StartStop	1	Running_Timer
Running_Wait	Timer	TEnd	Initial
Running_Wait	PositionSensor	0	Running_Timer_Reset
Running_Wait	ColorDetector	1	MotorUp
Running_Wait	StartStop	1	Running_Timer
Running_Wait	Abort	1	Aborted
Running_Timer_Reset	Tick	1	Running_Wait
Running_Timer_Reset	Abort	1	Aborted
MotorUp	PushButton	1	WhiteWait
MotorUp	StartStop	1	Motor_Up_Timer
MotorUp	Abort	1	Aborted
WhiteWait	StartStop	1	White_Wait_Timer
WhiteWait	Abort	1	Aborted
WhiteWait	Timer	SORT	MotorDown
MotorDown	StartStop	1	Motor_Down_Timer
MotorDown	Abort	1	Aborted
MotorDown	Timer	Motor Down	Running_Wait
Running_Timer	Timer	Tic	Running_Stop
Running_Timer	Abort	1	Aborted
Motor_Up_Timer	Timer	Tic	Motor_Up_Stop
Motor_Up_Timer	Abort	1	Aborted
White_Wait_Timer	Timer	Tic	White_Wait_Stop
White_Wait_Timer	Abort	1	Aborted
Motor_Down_Timer	Timer	Tic	Motor_Down_Stop
Motor_Down_Timer	Abort	1	Aborted
Motor_Up_Stop	PushButton	1	White_Wait_Stop
Motor_Up_Stop	Abort	1	Running_Stop
Motor_Up_Stop	Timer	Timer Interrupt	Initial
Motor_Up_Stop	Abort	1	Aborted
White_Wait_Stop	Timer	SORT	Motor_Down_Stop
White_Wait_Stop	Abort	1	Aborted
White_Wait_Stop	Timer	Timer Interrupt	Initial
Motor_Down_Stop	Timer	Motor Down	Running_Stop
Motor_Down_Stop	Abort	1	Aborted
Motor_Down_Stop	Timer	Timer Interrupt	Initial
Running_Stop	ColorDetector	1	Motor_Up_Stop
Running_Stop	Abort	1	Aborted
Running_Stop	Timer	Timer Interrupt	Initial
Aborted	StartStop	1	Initial

Validation of “State Transitions”

The description of our machine states is validated through its representation in the transition table. No state is excluded from being represented in the state transition table, all transitions will have the initial transition state differ from the end state.

Finite-state Automaton



Blue line means that the trigger for the transition is a clocktick

Validation of "Finite-state Automaton"

When we were making our finite-state automaton we looked at our state description and made sure that all states were represented, then we used our state transition table to make sure all transitions were correctly implemented.

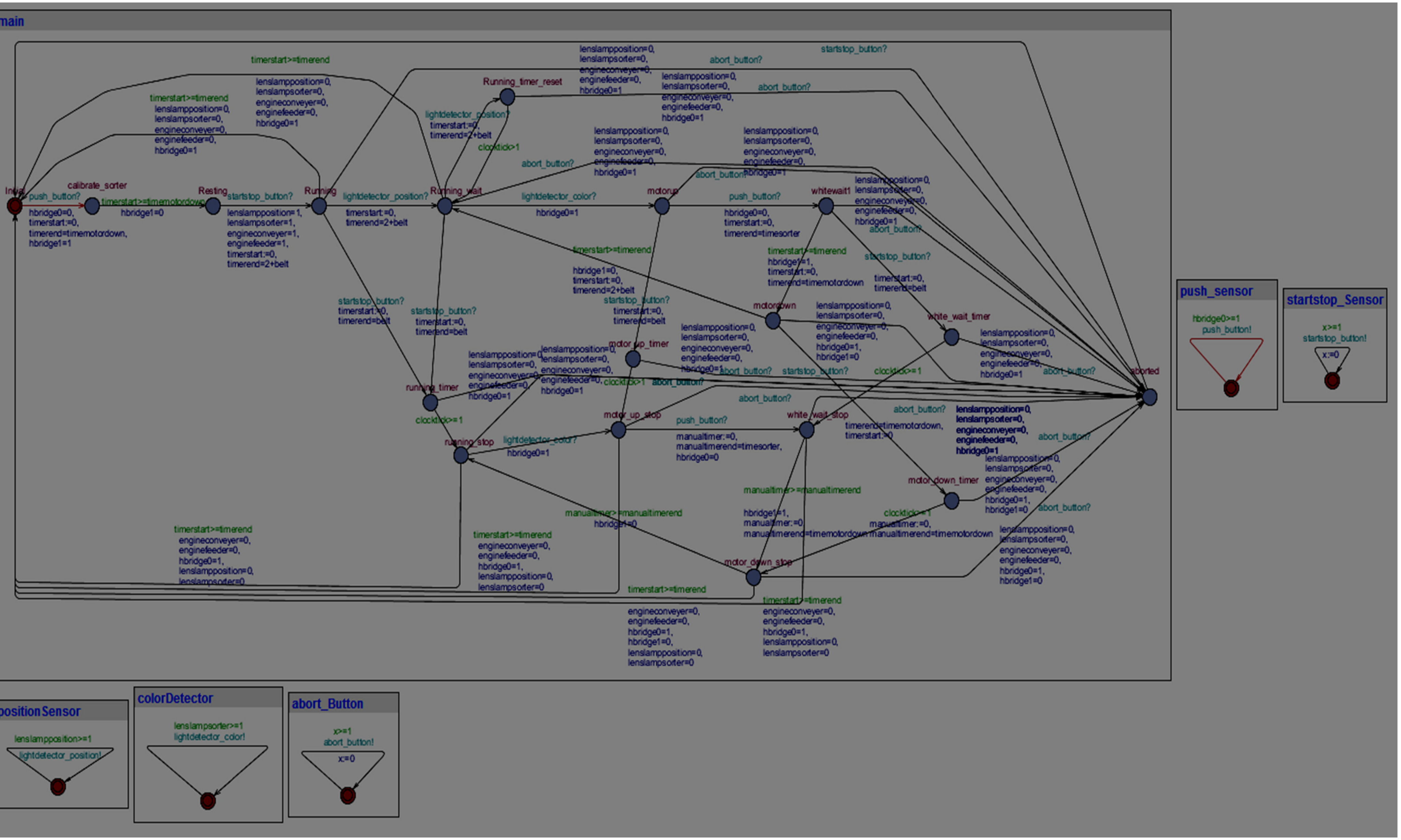
UPPAAL model

Tests done

On the next page is the UPPAAL model. This UPPAAL model has been tested for 2 safety properties. The first one is “ After the start-up of the machine, the assembly program should not stop until the machine is shut down.”. This has been tested using the following property “A[] not deadlock”, and we didn’t have a deadlock. The second safety property which was tested is: “The outputs connected to the h-bridge may never be powered on at the same time.”. This was tested using the following property “A<> !(hbridge0==1 && hbridge1=1)”. This one was also correct.

Validation of “UPPAAL model”

All transitions which exist in the UPPAAL model also occur in the Finite State Automaton. And the same action has to be performed to take that transition. Also all states of the Finite State Automaton occur in the UPPAAL model. The states of the UPPAAL model also have the outputs in them. The states of the Finite State Automaton do not have the outputs in them. Thus we validate the values of the outputs, which are in the states, to the description of the states.



main
Initial
calibrate_sorter
Resting
Running
Running_wait
Running_timer_reset
running_timer
running_stop
motorup
motor_up_timer
motor_up_stop
motordown
motor_down_timer
motor_down_stop
whitewait1
white_wait_timer
white_wait_stop
aborted
push_button?
startstop_button?
abort_button?
lightdetector_position?
lightdetector_color?
timerstart>=timerend
timerstart>=timemotordown
manualtimer>=manualtimerend
clocktick>1
clocktick>=1
push_sensor
hbridge0>=1
push_button!
startstop_Sensor
x>=1
startstop_button!
x:=0
positionSensor
lenslampposition>=1
lightdetector_position!
colorDetector
lenslampsorter>=1
lightdetector_color!
abort_Button
x>=1
abort_button!
x:=0

Software Design

In the Software Design phase, we present a Java program that realises the functions specified in the Software Specification document. This program is an intermediate step towards writing the PP2 code that controls the sorting machine.

Coding Standards

The java pseudo code follows the Google Java Style.
Source to Google Java Style: https://google-styleguide.googlecode.com/svn/trunk/javaguide.html.

PHP code used in this project follows the Zend Framework Coding Standard for PHP.
Source: http://framework.zend.com/manual/1.12/en/coding-standard.html.

Translating to pseudo java:

The java program starts by declaring the output variables. The names of the output variables will keep their original name, without spaces, in a camelCase form. The variable type will be determined from the Output table.

The inputs follow the same pattern.

Every state is represented as a function, keeping their name in the camelCase fashion, they will be all void functions due to the fact that they do not return anything.

Every state function will run preconditions if any, then check for specific input values using if statements, if an if statement is satisfied, there will be changes to the output values to match the next states output values, also the display is set to output the next states number, and then the next state function is called according to the state transition diagram, if no if statement is satisfied the current function is recalled.

The program is always looping, consequence of no deadlocks in the state machine as proven by the UPPAAL model test.

Example: Initial -> Calibrate_Sensor

So in this example the function initial is currently running, there are no preconditions to be checked, if the inputs have the desired value, in our case we check if the push button is pressed by the sorter, if so we will have the sorter moved down by activating the sorter motor via having the Hbridge0 variable set to 1. After this we set the display to showcase the number $branchTO where to branch to2 then call calibrateSensor function and if the if statement wasn't satisfied we recall initial entering a loop.

Translating from Java to PHP

The java code was written such that the conversion process to php is as easy as possible.

All variable in java will have the "$" sign added at the beginning of their name to comply with the php standards. The "$" sign has no influence in the java program variable naming, while in php it is mandatory.

Design decisions for the Java code

In translating our transition table to a Java program we made a number of decisions shaping the code, these decisions are outlined in this section.

We started by looking at our transition table, in this table we had our transitions ordered by the "current state", the state where the transition starts. Then there were some inputs that could trigger a transition from this state to a number of other states. Because of this we thought it would make sense to write a function for each state, since it would allow our code to essentially be a condensed version of the transition table. Where the code would be ordered by the "current state", and each state would have a number of outgoing transitions to other states. This resulted in the following blueprint for each of our functions:

```
void function(){
  timerManage();              // The function that manages the outputs and PWM
  $temp = getButtons();       // Store the buttons currently being pressed in a temporary
variable

  if( condition for transition ){         // Do this once for every outgoing transition from this
state
    Changes required to change to the new statement;
  }

  function();                 // Call on the same function again to recheck the buttons and
continue running the machine with timerManage()
}
```

Then we made an extra function which will be called from each function to do the PWM. This function is called timerManage. This function firstly gets the voltage which the output needs from the array.

This function has a variable called counter which increments each time the outputs have been set. That value is take modulo 12. So it will leave the outputs which need 12 volts on all the time. The reason why the values which need less than 12 volt will be turned off after they have been on for long enough. That goes as follows. First it checks if the engine needs to be on by checking if the voltage it needs is higher than counter. If the output needs to be on then it gets the location of the value in the array. And then does 2 to the power of the location. So now the correct output will be set on. Then the value of 2 to the power will be added to the variable engines. Then after all 7 outputs have been through that loop then it will set the output to the value of engines. So the lights which needed to be on will be on. Now the value of counter will increment each time and take modulo 12.

We also choose to save certain values, which may not be expected to be saved. In this section I will explain why we save the 2 variables. The first one is the variable of the location of the code. This has been saved because then we then we are capable of changing the return address after the timer interrupt. Because when an timer interrupt occurs we want to return to the initial state and the position where we were before. We also saved the original position of the stack pointer for when we come back from the timer interrupt to make sure that we empty the stack. Because there may be some values on the stack from before the timer interrupt. Thus to remove them we set the stack pointer to its original value.

Validation

Validation of java to transition table

Every state is represented by a function. The if statements in that function are the transitions which can occur from that state. The timer interrupt and the abort transitions are not represented as if statements, because interrupts go to a separate state(function). In those if statements the values that have to change are changed. The display will also be updated to the correct number of the state. The function timerManage is called in each state. Because with that function we make sure that the all outputs have the correct voltage.

We checked that all states are represented in the java code by a function. We also checked if they have all the transitions as if statements and that the correct values are changed.

Validation of timerManage

Loop invariant:

All elements before the current element of the array have been set on if they had to be on.

Initialize:

We start with the first element. Thus there are no elements before it and the loop invariant holds.

Step case:

If we're at element k, then according to the loop invariant all elements before k have been set on if they had to be on. Then if k has to be on (value of k>counter) it will be set on else it will stay off. So now the loop invariant holds for the element k+1

Termination:

The loop will terminate when k is greater than 7. Because we do not have any more outputs.

Control flow validation

Because the Java code has been validated to the state description and the transition table, which, in turn, have been validated with the UPPAAL model and shown to be correct and in tune with the initial description of the sorting machine. This means that the Java program, being a one-to-one translation of the finite state automaton, also has a correct control flow.

Software Implementation and Integration

Now we show the data representation and coding standard we chose that is used to write the Assembly Language.

Java to PHP

The Java to PHP conversion is usually natural, the two languages sharing most syntax but there are some differences we must note down. We are not required to create a class in PHP. The initialization will differ in PHP from Java, but they share the same core in the end. Also while we have some of the variables initialized globally in Java, in PHP they will be local. Having no class will make the class initialization irrelevant in PHP and that's why its missing. The later functions in the Java code right after the function TimerManage are included in the PHP code using "include "functions.php";". In TimerManage, % operation is replaced by the mod() function. Due to our PHP compiler limitations we are required to use variables as arguments when calling certain functions like for example storeData. The PHP code has been added as appendix 4.

Validation of PHP to Java

Because of the natural similarity and ease of conversion, the PHP codes correctness can be correlated to its java counterpart, the correctness of the java code was validated in the Validation part of the Software Design.

Validation of Assembly to PHP compiler

The compiler works in phases. We will go through these phases 1 by 1 to explain how the compiler does its job: compiling PHP-like code to assembly. Throughout the phases the compiler keeps track of the line number of the PHP code it is currently compiling and uses that, when an error occurs, to give information where the error is. The compiler is written in PHP5.6 and uses a command line interface.

Preprocessing

In the first phase, the input code will be made ready for the next steps. A few things happen in this phase: First the file is read into the memory. The next step is that all comments, newlines and extra spaces are stripped from the file. The file is then split into single lines using the ";" symbol that denotes the end of a line. While doing this the compiler writes the data to two arrays: the data array for everything between "//**DATA**" and "//**CODE**" and the code array for everything after "//**CODE**". Everything before //**DATA** is ignored. The data array gets compiled immediately.

The preprocessor further removes some special statements that are needed to make valid php such as "global" and changes some shortcuts in their full version. For example $abc++ will be changed into $abc+=1. This ensures that the compiler only needs to be able to handle $abc+=1.

Splitting

In the second phase the code is split up by function. Every function gets his own array with all the lines that are in that function. The code not inside of a function goes into a separate array.

Compiling
The third phase is the most important one. It starts by compiling the code that is at the start and not inside a function. While compiling it keeps track of what functions are called and adds these, if they are not already compiled, to the toCompile queue. This helps in making sure there is no dead code, as a function that is never called, will not be compiled. The compiler adds the function “main”, which is the default start point of the code, to the queue and starts processing it.

After compiling the main function it will continue in the next function in the toCompile queue and keep doing this till the toCompile queue is empty.

The compiling itself is not a lot more than a lot of regex and switch statements that look at the input and make an output from that. At the first notion of a variable a register is assigned to it. The code then uses this register in place of the variable. Some more difficult statements, like the function display which displays something, will BRS to premade assembly code that handles that. The compiler keeps track of which segments of the premade assembly code are used.

When the compiler meets an if statement, it saves the code inside it to a new function named “condtionali” where i is the amount of conditionals that have already been seen. It then places this function in the toCompile queue. It also saves the location of the end of the if statement, so it will later know where to return when the if function has ended.

Combining
After there are no functions left in the toCompile queue, the combining phase starts. In this phase all the functions and the code outside the functions are combined into a single array. This phase also adds the used premade functions at the top and inserts the return statements at the correct position.

Formatting
The last phase is the last interesting. It goes through the, now compiled code, and formats it. It uses either the length of the longest function name or the number 25 depending on which is larger to insert spaces in front of every line of code in a way everything lines up nicely.

The last step the compiler takes is writing the compiled code to a file and using the assembler provided to create the hex code.

System Validation and Testing

Finally, we demonstrate that the final product meets its initial requirements, i.e. we prove that the executable code correctly implements the System Level Requirements, and that the implementation doesn't do more than is expected.

Validation Policy

In our documents we have validated every element of contents in a separate Validation section at the end of the document or near to it.

Machine Design will have at the end of the document a Validation section which includes the Validation of High Level Specifications and the Validation of the System Level Requirements, also adding Validation to Design Priorities.

The Software Specification document will have a Validation section that will contain the validation of the Inputs and Outputs, the Relation of Inputs and Outputs, the Description of States, the State Transitions, the Finite State Automaton and the UPPAAL model.

Software Design will have a Validation section close to the end of the document being afterwards followed by the Program Code. The Validation will contain the validation of the java code to the transition table(from the Software Specification), validation of the timerMange function (this function needed separate formal proof for its inner loop) and Control flow validation.

The Software Implementation and Integration document will have at the end a Validation section containing validation of the PHP code to java and the validation of the Assembly code to the PHP compiler.

Conclusion

To be completed.

Product to Problem

To be completed.

Process

Design decisions

Work Plan
To streamline the group process we needed a Work Plan. We started this Work Plan with the inventory of the goals and objectives of each phase of the project. For the roles in the group we chose to have them the same as described in /Project Guide Design Based Learning "DBL 2IO70" "Sort It Out"/.

Then we come to the definition of our terms. We chose to have abbreviations of the phases and the tasks. This way we can refer to them without having to waste a lot of space if we mention them multiple times. Also the roles have their abbreviations.

Before we use those abbreviations we first have an inventory of the amount of work and an overview of the main deliverables. The amount of work is given per phase and week in a nifty table. The overview of deliverables contains who's responsible for a certain deliverable and the date and week the deliverable is due.

Then we come to the weekly tables. Tuesday and Friday we have a tutor meeting and we work afterwards till in the afternoon. On Wednesday we have Data Structures in the morning and work on the project afterwards. Those times are included in the tables. Everyone has his column with his role if applicable. For every hour and person it's defined what he will be working on.

With this Work Plan and the collective logbook we're able to have an indication of how much time was spent on each task by each member. If necessary action can be taken based on this indication.

If unforeseen problems arise and the deadline is close, this means we have to work harder. Deadlines aren't easily moved. If someone spends too less time according to the Work Plan it's expected he does his work at home.

Workday
For us, a normal workday is structured as follows: we start each workday with a list of items that needs to be done in order to complete the document for that week. The list is written on the whiteboard that is available in the room. Then members are assigned to a task in consultation. After the completion of a task, it is checked off or removed from the whiteboard, and the member that was responsible for it continues to work on the next item of the inventory until there are no more available assignments. Next, they will help another group member with their duty. This cycle repeats itself whenever we are together. On Wednesday, the document is wrapped up and cross-read. The person that bears the responsibility for the document hands in the current document for feedback when possible. On Friday, the document is updated according to the feedback given by the tutor. Subsequently, the finalised document is cross-read, and handed in by the person responsible for the document.

Problems
There was a problem with the group not functioning as was expected. The logbook indicated that some members contributed less than other members. As a result, other members had to compensate for it by spending more time on the project. Therefore, we decided to address this problem in the meetings and to distribute the workload more evenly.

Conclusion

Over the course of these past 8 weeks we worked on making a sorting machine and the software that runs it. We did this by going through multiple phases, starting with Machine Design, where we designed the machine itself. Moving to Software Specification, where we created a finite state automaton, then Software Design and Software Implementation and Integration where we respectively designed a pseudo-Java program and then translated that into Assembly for the PP2. While making these documents we validated each part to what we did before to make sure that we made the right decision every time. While the project took a lot of our time each week, we liked doing it, and the end result was very satisfying. We hope that the skills we have acquired over the course of this project, both those for designing and building a product and those for working in a group, will help us in future projects both here in the TU/e and beyond.

Appendix 1: Java Program

```java
/**
 * Sort of a simulation of the PP2 program
 * controlling the Fischer
 * Technik in order to sort black and white discs.
 *
 * @author Maarten Keet
 * @author Stefan van den Berg
 * @author Rolf Verschuuren
 * @author Wigger Boelens
 * @team Group 16
 * @since 13/3/2015
 */


class SoftwareDesign {
    //**@CODE**
    //inputs
    int $push, $startStop, $abort, $position,
            $colour;

    //variables
    int $state = 0;
    int $sleep = 0;
    int $temp = 0;
    int $location;
    int $counter = 0;
    int $engines;


    //constants
    final int TIMEMOTORDOWN = 30;
    final int BELTROUND = 2000;
    final int BELT = 1200;
    final int SORT = 850;
    final int LENSLAMPPOSITION = 5,
            LENSLAMPSORTER = 6,
            HBRIDGE0 = 0,
            HBRIDGE1 = 1,
            CONVEYORBELT = 3,
            FEEDERENGINE = 7,
            DISPLAY = 8,
            LEDSTATEINDICATOR = 9;

    public static void main(String args[]) {
        SoftwareDesign SoftwareDesign = new
                SoftwareDesign();


        //values for the data segment
        SoftwareDesign.initVar("outputs", 12);
        SoftwareDesign.initVar("stackpointer", 1);
        SoftwareDesign.initVar("offset", 1);

        //store the offset of the programm,this
        // is used in the interrupt
        SoftwareDesign.storeData(startofthecode,
                                 "offset", 0);

        //store the vlue of the stackpointer,so
        // we can clear the stack
        // easily
        SoftwareDesign.storeData(SP,
                                 "stackpointer",
                                 0);

        $counter = 0;


        //reset outputs
        SoftwareDesign.storeData(0, "outputs",
                                 SoftwareDesign
                                .HBRIDGE1);
        SoftwareDesign.storeData(0, "outputs",
                                 SoftwareDesign
                                .LENSLAMPPOSITION);
        SoftwareDesign.storeData(0, "outputs",
                                 SoftwareDesign
                                .LENSLAMPSORTER);
        SoftwareDesign.storeData(0, "outputs",
                                 SoftwareDesign
                                .LEDSTATEINDICATOR);
        SoftwareDesign.storeData(0, "outputs",
                                 SoftwareDesign
                                .DISPLAY);
        SoftwareDesign.storeData(0, "outputs",
                                 SoftwareDesign
                                .CONVEYORBELT);
        SoftwareDesign.storeData(0, "outputs",
                                 SoftwareDesign
                                .FEEDERENGINE);

        //start moving the sorter up
        SoftwareDesign.storeData(9, "outputs",
                                 SoftwareDesign
                                 .HBRIDGE0);


        //go to the first state and set the
        // value for the display
        SoftwareDesign.$state = 0;
        SoftwareDesign.initial();
    }

    //state0
    void initial() {
        setStackPointer(
                getData("stackpointer", 0));
        timerManage();
        //check if the sorter push button is
        // pressed
        $push = getButtonPressed(5);
        if ($push == 1) {
            //move the sorter down
            storeData(0, "outputs", HBRIDGE0);
            storeData(9, "outputs", HBRIDGE1);
            //update the state
            $state = 1;
            //reset sleep for the next function
            $sleep = 0;
            calibrateSorter();

        }
        //loop
        initial();
    }

    //state 1
    void calibrateSorter() {
        timerManage();
        //the sorter is now moving down,
        //and we're waitng for it to reach the
        // bottom
        if ($sleep == TIMEMOTORDOWN * 1000) {
            //stop the sorter
            storeData(0, "outputs", HBRIDGE1);
            //update the state
            $state = 2;
            //reset sleep
            $sleep = 0;
            resting();
        }
        //loop
        $sleep++;
        calibrateSorter();
    }

    //state 2
    void resting() {
        timerManage();
        //the program waits for the user to
        // press the start/stop
        $startStop = getButtonPressed(0);
        if ($startStop == 1) {
            //sleep so we don't go to the pause
            // immediatly
            sleep(2000);
            //power up the lights
            storeData(12, "outputs",
                      LENSLAMPPOSITION);
            storeData(12, "outputs",
                      LENSLAMPSORTER);
            //start up the belt and the feeder
            storeData(9, "outputs", CONVEYORBELT);
            storeData(5, "outputs", FEEDERENGINE);
            //set and start the countdown
            setCountdown(BELTROUND + BELT);
            startCountdown();
            //update the state
            $state = 3;
            running();
        }
        //loop
        resting();
    }

    //state 3
    void running() {
        timerManage();
        //check if we need to pause
        $startStop = getButtonPressed(0);
        if ($startStop == 1) {
            //stop the feeder engine
            storeData(0, "outputs", FEEDERENGINE);
            //set the timer
            setCountdown(BELT);
            //update the state
            $state = 9;
            runningTimer();
        }
        //check if a disk is at the position
        // detector
        $position = getButtonPressed(7);
        if ($position == 1) {
            //reset the countdown,because a
            // disk was detected
            setCountdown(BELTROUND + BELT);
            //update the state
            $state = 4;
            runningWait();
        }
        //loop
        running();
    }
```

```

void runningWait() {
    timerManage();
    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        storeData(0, "outputs", FEEDERENGINE);
        //set the timer
        setCountdown(BELT);
        //update the state
        $state = 9;
        runningTimer();
    }
    //check if a disk is at the positiond
    // detector
    $position = getButtonPressed(7);
    if ($position == 1) {
        //reset the countdown,because a
        // disk was detected
        setCountdown(BELTROUND + BELT);
        //update the state
        $state = 5;
        runningTimerReset();
    }
    //check if a white disk is at the color
    // detector
    $colour = getButtonPressed(6);
    if ($colour == 1) {
        //move the sorter up
        storeData(9, "outputs", HBRIDGE0);
        //update the state
        $state = 6;
        motorUp();
    }
    //loop
    runningWait();
}

//state 5
void runningTimerReset() {
    timerManage();
    //update the state
    $state = 5;
    runningWait();
}

//state 6
void motorUp() {
    timerManage();
    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        storeData(0, "outputs", FEEDERENGINE);
        //set the timer
        setCountdown(BELT);
        motorUpTimer();
    }
    //check if the sorter push button is
    // pressed
    $push = getButtonPressed(5);
    if ($push == 1) {
        //stop the engine,because it is in
        // the right position
        storeData(0, "outputs", HBRIDGE0);
        //update the state
        $state = 7;
        whiteWait();
    }
    //loop
    motorUp();
}

//state 7
void whiteWait() {
    timerManage();
    //we are waiting for the white disk to
    // be sorted
    if ($sleep == SORT * 1000) {
        //start moving the sorter down
        storeData(9, "outputs", HBRIDGE1);
        //update the state
        $state = 8;
        //reset sleep for the next function
        $sleep = 0;
        motorDown();

    }
    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        storeData(0, "outputs", FEEDERENGINE);
        //set the timer
        setCountdown(BELT);
        //update the state
        $state = 11;
        whiteWaitTimer();
    }
    //loop
    $sleep++;
    whiteWait();
}

//state 8
void motorDown() {
    timerManage();
    //the sorter is moving down
    if ($sleep == TIMEMOTORDOWN * 1000) {
        //stop the sorter
        storeData(0, "outputs", HBRIDGE1);
        //update the state
        $state = 9;
        //reset sleep for the next function
        $sleep = 0;
        runningWait();
    }
    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        storeData(0, "outputs", FEEDERENGINE);
        //set the timer
        setCountdown(BELT);
        motorDownTimer();
    }
    //loop
    $sleep++;
    motorDown();

}

//state 9
void runningTimer() {
    timerManage();
    //update state
    $state = 13;
    runningStop();
}

//state 10
void motorUpTimer() {
    timerManage();
    //update state
    $state = 14;
    motorUpStop();
}

//state 11
void whiteWaitTimer() {
    timerManage();
    //update state
    $state = 15;
    whiteWaitStop();
}

//state 12
void motorDownTimer() {
    timerManage();
    //update state
    $state = 16;
    motorDownStop();
}

//state 13
void runningStop() {
    timerManage();
    //check if a white disk is at the
    // colour detector
    $colour = getButtonPressed(6);
    if ($colour == 1) {
        //move the sorter engine up
        storeData(9, "outputs", HBRIDGE0);
        //update the state
        $state = 10;
        motorUpStop();
    }
    //loop
    runningStop();
}

//state 14
void motorUpStop() {
    timerManage();
    //check if the sorter push button is
    // pressed
    $push = getButtonPressed(5);
    if ($push == 1) {
        //stop the engien for the sorter
        storeData(0, "outputs", HBRIDGE0);
        //update the state
        $state = 11;
        whiteWaitStop();
    }
    motorUpStop();
}

//state 15
void whiteWaitStop() {
    timerManage();
    //check if the white disk has been sorted
    if ($sleep == SORT * 1000) {
        //start moving the sorter down
        storeData(9, "outputs", HBRIDGE1);
        //update the state
        $state = 12;
        //reset the sleep for the next
        // function
        $sleep = 0;
        motorDown();
    }
    //loop
    $sleep++;
    whiteWaitStop();
}
```

```

    //state 16
    void motorDownStop() {
        timerManage();
        //check if the sorter has moved down
        if ($sleep == TIMEMOTORDOWN) {
            //stop the engine of the sorter
            storeData(0, "outputs", HBRIDGE1);
            //update the state
            $state = 9;
            //reset sleep for the next function
            $sleep = 0;
            runningWait();
        }
        //loop
        $sleep++;
        motorDownStop();
    }

    //not a state
    void timerInterrupt() {
        //show that we have timer interrupt
        $state = 18;
        //make the sorter move up
        storeData(9, "outputs", HBRIDGE0);
        //stop all other outputs
        storeData(0, "outputs", HBRIDGE1);
        storeData(0, "outputs", LENSLAMPPOSITION);
        storeData(0, "outputs", LENSLAMPSORTER);
        storeData(0, "outputs",
                  LEDSTATEINDICATOR);
        storeData(0, "outputs", DISPLAY);
        storeData(0, "outputs", CONVEYORBELT);
        storeData(0, "outputs", FEEDERENGINE);
        //make sure that the outputs get set
        // immediatly
        timerManage();
        //set the display to the state of initial
        $state = 0;

        initial();

    }

    void abort() {
        //stop all outputs
        storeData(0, "outputs", HBRIDGE0);
        storeData(0, "outputs", HBRIDGE1);
        storeData(0, "outputs", LENSLAMPPOSITION);
        storeData(0, "outputs", LENSLAMPSORTER);
        storeData(0, "outputs",
                  LEDSTATEINDICATOR);
        storeData(0, "outputs", DISPLAY);
        storeData(0, "outputs", CONVEYORBELT);
        storeData(0, "outputs", FEEDERENGINE);
        //make sure the outputs stop immediatly
        timerManage();
        //update the state to be correct in
        // aborted
        $state = 17;
        aborted();

    }

    //state 17
    void aborted() {
        timerManage();
        //check if we can start again
        $startStop = getButtonPressed(0);
        if ($startStop == 1) {
            //start moving the sorter up for
            // calibration
            storeData(1, "outputs", HBRIDGE0);
            //update the state
            $state = 0;
            initial();
        }
        //loop
        aborted();

    }

    void timerManage() {


        //make sure that when counter can not
        // be higher than 12
        mod(13, $counter);
        //get the voltage of output $location
        int $voltage = getData("outputs",
                               $location);
        //power up the output when it needs to
        if ($voltage > $counter) {
            $engines += pow(2, $voltage);
        }
        //check if we are in a new itteration
        if ($counter == 0) {
            //set the first part of the display
            $temp = getData("state", 0);
            mod(10, $temp);
            display($temp, "display", "1");


        }
        //check if we are at the end of the
        // itteration
        if ($counter == 12) {
            //set the second part of the display;
            $temp = getData("state", 0);
            $temp = $temp / 10;
            mod(10, $temp);
            display($temp, "display", "01");

        }
        //check if we did all outputs
        if ($location > 7) {
            display($engines, "leds", "");
            //set the variables for the next run
            $engines = 0;
            $location = 0;
            $counter++;

            //check if abort is pressed
            $abort = getButtonPressed(1);
            if ($abort == 1) {
                abort();//stop the machine
            }
            return;
        }


        $location++;
        timerManage();
    }
}
```

Appendix 2: Explanation of the compiler

The compiler works in phases. We will go through these phases 1 by 1 to explain how the compiler does its job: compiling PHP-like code to assembly. Throughout the phases the compiler keeps track of the line number of the PHP code it is currently compiling and uses that, when an error occurs, to give information where the error is. The compiler is written in PHP5.6 and uses a command line interface.

Preprocessing

In the first phase, the input code will be made ready for the next steps. A few things happen in this phase: First the file is read into the memory. The next step is that all comments, newlines and extra spaces are stripped from the file. The file is then split into single lines using the ";" symbol that denotes the end of a line. The code is divided in three segments. The first segment starts at //**COMPILER, everything before this statement is ignored.

The preprocessor further removes some special statements that are needed to make valid php such as "global" and changes some shortcuts in their full version. For example $abc++ will be changed into $abc+=1. This ensures that the compiler only needs to be able to handle $abc+=1.

Splitting

In the second phase the code is split up by function. Every function gets his own array with all the lines that are in that function. The code not inside of a function goes into a separate array.

Compiling

The third phase is the most important one. It starts by compiling the code that is at the start and not inside a function. While compiling it keeps track of what functions are called and adds these, if they are not already compiled, to the toCompile queue. This helps in making sure there is no dead code, as a function that is never called, will not be compiled. The compiler adds the function "main", which is the default start point of the code, to the queue and starts processing it.

After compiling the main function it will continue in the next function in the toCompile queue and keep doing this till the toCompile queue is empty.

The compiling itself is not a lot more than a lot of regex and switch statements that look at the input and make a output from that. At the first notion of a variable a register is assigned to it. The code then uses this register in place of the variable. Some more difficult statements, like the function display which displays something, will BRS to premade assembly code that handles that. The compiler keeps track of which segments of the premade assembly code are used.

When the compiler meets an if statement, it saves the code inside it to a new function named "condtionali" where i is the amount of conditionals that have already been seen. It then places this function in the toCompile queue. It also saves the location of the end of the if statement, so it will later know where to return when the if function has ended.

For every line it compiles, it takes the corresponding line of PHP and inserts it as a comment in the assembly. This is to help in debugging.

Combining

After there are no functions left in the toCompile queue, the combining phase starts. In this phase all the functions and the code outside the functions are combined into a single array. This phase also adds the used premade functions at the top and inserts the return statements at the correct position.

Formatting

The last phase is the least interesting. It goes through the, now compiled code, and formats it. It uses either the length of the longest function name or the number 25 depending on which is larger to insert spaces in front of every line of code in a way everything lines up nicely. It also makes sure the comments line up nicely.

The last step the compiler takes is writing the compiled code to a file and using the assembler provided to create the hex code.

Appendix 3: Explanation of the compiler functions

storeRam($location, $value)

Store a value in the ram.

$location	The location (a variable) to store the value in the ram
$value	The value to store, needs to be a variable
return	void

getRam($location)

Get a value from the ram.

$location	The location (a variable) where the value is stored
return	The value that is stored at the location

display($what, $onWhat, $location = '000001')

Display something on either the display or the leds.

Possible values for $onwhat:

- leds: the leds at the top
- leds2: the leds to the right
- display: the display

$what	What to display, must be a variable
$onWhat	On what to display
$location	Where to show the value when using the display, defaults to the right position
return	void

pow($number,$power)

Get the power of a number

$number	The number to power
$power	The power value
return	Int; The result

mod($what, $variable)

Take the modulo of a number

$what	Modulo what

$variable	Variable to modulo over
return	void

getInput($writeTo, $type)

Get button or analog input. When you just want the input of 1 button, use getButtonPressed instead.

$writeTo	Variable to write the input to
$type	Type of input, possible values are: buttons, analog
return	void

getButtonPressed($button)

Check if a button is pressed. Puts the result into R5.

$button	Which button to check (input a variable)
return	Int; Whether or not the button is pressed.

installCountdown($functionName)

Install the countdown.

$functionName	The name of the function where the timer should go to
return	void

startCountdown()

Start the countdown.

Retrun	void

pushStack($variable)

Push a variable to the stack

$variable	The variable to push to the stack
return	void

pullStack($variable)

Pull a variable from the stack.

$variable	The variable where the pulled variable is put into
return	void

setCountdown($countdown)

Set the timer interrupt to a value. It will first reset the timer to 0.

$countdown	How long the countdown should wait, in timer ticks
return	void

getData($location, $offset)

Get data. Use offset 0 when it is just a single value.

$location	The location where the variable is stored
$offset	The offset of the location
return	The value of the data segment

storeData($variable, $location, $offset)

Store data. Use offset 0 when it is just a single value.

$variable	The variable to store
$location	The name of the location where the variable is stored
$offset	The offset of the location
return	void

sleep($howLong)

Pause the program.

$howLong	How long to sleep in clockticks
return	void

initVar($variable,$places)

Initialize a variable that is used in that data segment.

$variable	The name of the variable
$places	How long the array is
return	void

branch($branchTO)

Branch to a function.

$branchTO	where to branch to
return	void

moveFunction($branchTO)

Move a function in the assembly code.

$branchTO	Where to branch to

return void

Appendix 4: PHP Program

```php
<?php
/* vim: set expandtab tabstop=4 shiftwidth=4 softtabstop=4: */

/**
 * Sort of a simulation of the PP2 program controlling the Fischer Technik in order to sort black and white discs.
 * @team Group 16
 * @author Stefan van den Berg
 * @author Rolf Verschuuren
 * @author Wigger Boelens
 * @since 13/3/2015
 */
include 'functions.php';
//**COMPILER**
moveFunction('timerInterrupt', 1);
moveFunction('timerManage', 50);

//**DATA**
initVar('offset', 1);
initVar('stackPointer', 1);
initVar('outputs', 12);
initVar('state', 1);

//**CODE**
define('TIMEMOTORDOWN', 150); //how long the sorter takes to move down
define('BELT', 2000);
define('BELTROUND', 2000);//Time for the belt to make a rotation
define('SORT', 200);//Clockticks to make a rotation
define('COUNTDOWN', 30000);
//outputs
define('LENSLAMPPOSITION', 2);
define('LENSLAMPSORTER', 6);
define('HBRIDGE0', 0);
define('HBRIDGE1', 1);
define('CONVEYORBELT', 7);
define('FEEDERENGINE', 3);
define('DISPLAY', 8);
define('LEDSTATEINDICATOR', 9);

//not a state
function main()
{
    global $counter, $location;

    //store the offset of the program, this is used in the interrupt
    storeData(R5, 'offset', 0);
    //install the countdown
    installCountdown('timerInterrupt');

    //save the location of the stackPointer, so we can clear the stack
    storeData(SP, 'stackPointer', 0);

    //the variables that are the same throughout the program:
    $counter = 0;
    $location = 0;
    $sleep = 0;


    //stop everything
    $temp = 0;
    storeData($temp, 'outputs', HBRIDGE1);
    storeData($temp, 'outputs', LENSLAMPPOSITION);
    storeData($temp, 'outputs', LENSLAMPSORTER);
    storeData($temp, 'outputs', LEDSTATEINDICATOR);
    storeData($temp, 'outputs', DISPLAY);
    storeData($temp, 'outputs', CONVEYORBELT);
    storeData($temp, 'outputs', FEEDERENGINE);

    //sh0w the state
    $state = 0;
    storeData($state, 'state', 0);

    //set HBridge so the sorter starts moving up
    $temp = 10;
    storeData($temp, 'outputs', HBRIDGE0);
    unset($temp, $state);

    //go to the first state
    initial();
}

//state 0
function initial()
{
    global $sleep;
    //disable the lights on the right hand side
    $temp = 0;
    display($temp, 'leds2');

    $temp = getData('stackPointer', 0);
    setStackPointer($temp);

    timerManage();

    //check if the sorter push button is pressed
    $push = getButtonPressed(5);
    if ($push == 1) {
        //move sorter down
        $temp = 0;
        storeData($temp, 'outputs', HBRIDGE0);
        $temp = 10;
        storeData($temp, 'outputs', HBRIDGE1);

        //update state
        $temp = 1;
        storeData($temp, 'state', 0);
        unset($temp);

        //reset sleep for the next function
        $sleep = 0;
        calibrateSorter();

    }
    unset($push);

    //loop
    initial();
}

//state 1
function calibrateSorter()
{
    global $sleep;
    timerManage();

    //the sorter is now moving down,
    //we're waiting for it to reach its bottom position
    if ($sleep == TIMEMOTORDOWN) {
        //stop the sorter
        $temp = 0;
        storeData($temp, 'outputs', HBRIDGE1);

        //update the state
        $state = 2;
        storeData($state, 'state', 0);
        unset($state);

        //reset sleep for the next state
        $sleep = 0;
        resting();
    }

    //loop
    $sleep++;
    calibrateSorter();
}

//state 2
function resting()
{
    timerManage();

    //the program is now waiting for the user to press start/stop
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //sleep so we don't go to pause immediately


        //power up the lamps
        $temp = 12;
        storeData($temp, 'outputs', LENSLAMPPOSITION);
        unset($temp);
        timerManage();
        sleep(1000);
        $temp = 12;
        storeData($temp, 'outputs', LENSLAMPSORTER);
        unset($temp);
        timerManage();
        sleep(2000);


        //start up the belt and feeder
        $temp = 9;
        storeData($temp, 'outputs', CONVEYORBELT);
        $temp = 9;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //set and start the countdown for the moment there are no more disks
        //this countdown will reset every time a disk is found
        //when it triggers, timerInterrupt will be ran.
        setCountdown(COUNTDOWN);
        startCountdown();

        //update the state
        $state = 3;
        storeData($state, 'state', 0);
        unset($state);

        running();
    }
    unset($startStop);

    //loop
    resting();
}

//state 3
function running()
{
    timerManage();

    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //exit after 1 rotation of the belt
        setCountdown(BELT * 10);

        //update the state
        $state = 9;//TODO: echte state
        storeData($state, 'state', 0);
        unset($state);

        runningTimer();

    }
    unset($startStop);

    //check if a disk is at the position detector
    $position = getButtonPressed(7);
    if ($position == 1) {
        //reset the countdown, because a disk was just detected
        setCountdown(COUNTDOWN);

        //update the state
        $state = 4;
        storeData($state, 'state', 0);
        unset($state);
        runningWait();
    }
    unset($position);

    //loop
    running();
}

//state 4
function runningWait()
{
    timerManage();

    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //exit after 1 rotation of the belt
        setCountdown(BELT * 10);

        //update the state
        $state = 9;
        storeData($state, 'state', 0);
        unset($state);

        runningTimer();

    }
    unset($startStop);

    //check if a disk is at the position detector
    $position = getButtonPressed(7);
    if ($position == 0) {
        //reset the countdown, because a disk was just detected
        setCountdown(COUNTDOWN);

        //update state
        $state = 5;
        storeData($state, 'state', 0);
        unset($state);

        runningTimerReset();

    }
    unset($position);

    //check if a white disk is at the colour detector
    $colour = getButtonPressed(6);
    if ($colour == 1) {
        //move the sorter up so the disk goes to the correct box
        $temp = 10;
        storeData($temp, 'outputs', HBRIDGE0);

        //stop the feeder engine
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //update state
        $state = 6;
        storeData($state, 'state', 0);
        unset($state);

        motorUp();
    }
    unset($colour);

    //loop
    runningWait();
}

//state 5
function runningTimerReset()
{
    timerManage();

    //update state
    $state = 4;
    storeData($state, 'state', 0);
    unset($state);

    runningWait();
}

//state 6
function motorUp()
{
    global $sleep;
    timerManage();

    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //exit after 1 rotation of the belt
        setCountdown(BELT * 10);

        //update the state
        $state = 10;
        storeData($state, 'state', 0);
        unset($state);

        motorUpTimer();

    }
    unset($startStop);

    //check if the sorter push button is pressed
    $push = getButtonPressed(5);
    if ($push == 1) {
        //stop the sorter engine, because its at its highest position
        $temp = 0;
        storeData($temp, 'outputs', HBRIDGE0);
        unset($temp);

        //update state
        $state = 7;
        storeData($state, 'state', 0);
        unset($state);

        //set sleep for the next function
        $sleep = 0;

        whiteWait();
    }
    unset($push);

    //loop
    motorUp();
}

//state 7
function whiteWait()
{
    global $sleep;
    timerManage();

    //we are waiting for the white disk to be sorted
    if ($sleep == SORT) {
        //start moving the sorter down
        $temp = 10;
        storeData($temp, 'outputs', HBRIDGE1);
        unset($temp);

        //make sure the timerinterrupt is correct
        setCountdown(COUNTDOWN);

        //update state
        $state = 8;
        storeData($state, 'state', 0);
        unset($state);

        //reset sleep for the next function
        $sleep = 0;
        motorDown();

    }

    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //exit after 1 rotation of the belt
        setCountdown(BELT * 10);

        //update the state
        $state = 11;
        storeData($state, 'state', 0);
        unset($state);

        whiteWaitTimer();
    }
    unset($startStop);

    //loop
    $sleep++;
    whiteWait();
}

//state 8
function motorDown()
{
    global $sleep;
    timerManage();


    //check if a white disk is at the colour detector
    $colour = getButtonPressed(6);
    if ($colour == 1) {
        //move the sorter up so the disk goes to the correct box
        $temp=0;
        storeData($temp,'outputs',HBRIDGE1);
        $temp = 10;
        storeData($temp, 'outputs', HBRIDGE0);
        unset($temp);

        //update state
        $state = 6;
        storeData($state, 'state', 0);
        $sleep=0;
        unset($state);

        motorUp();
    }
    unset($colour);


    //the sorter is moving down, we are waiting for that to complete
    if ($sleep == TIMEMOTORDOWN) {
        //stop the sorter, its where it should be
        $temp = 0;
        storeData($temp, 'outputs', HBRIDGE1);
        $temp = 7;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //update state
        $state = 4;
        storeData($state, 'state', 0);
        //reset sleep for the next function
        $sleep = 0;
        unset($state);

        runningWait();
    }

    //check if we need to pause
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //stop the feeder engine
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //exit after 1 rotation of the belt
        setCountdown(BELT * 10);

        //update the state
        $state = 12;
        storeData($state, 'state', 0);
        unset($state);

        motorDownTimer();
    }
    unset($startStop);

    //loop
    $sleep++;
    motorDown();

}

//state 9
function runningTimer()
{
    timerManage();

    //update state
    $state = 13;
    storeData($state, 'state', 0);
    unset($state);

    runningStop();
}

//state 10
function motorUpTimer()
{
    timerManage();

    //update state
    $state = 14;
    storeData($state, 'state', 0);
    unset($state);

    motorUpStop();
}

//state 11
function whiteWaitTimer()
{
    timerManage();

    //update state
    $state = 15;
    storeData($state, 'state', 0);
    unset($state);

    whiteWaitStop();
}

//state 12
function motorDownTimer()
{
    timerManage();

    //update state
    $state = 16;
    storeData($state, 'state', 0);
    unset($state);

    motorDownStop();
}

//state 13
function runningStop()
{
    timerManage();

    //check if a white disk is at the colour detector
    $colour = getButtonPressed(6);
    if ($colour == 1) {
        //stop the sorter engine, because its at its highest position
        $temp = 10;
        storeData($temp, 'outputs', HBRIDGE0);

        //stop the feeder engine
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //update state
        $state = 10;
        storeData($state, 'state', 0);
        unset($state);

        motorUpStop();
    }
    unset($colour);

    //loop
    runningStop();
}

//state 14
function motorUpStop()
{
    timerManage();

    //check if the sorter push button is pressed
    $push = getButtonPressed(5);
    if ($push == 1) {
        //stop the engine of the sorter
        $temp = 0;
        storeData($temp, 'outputs', HBRIDGE0);
        unset($temp);

        //update state
        $state = 11;
        storeData($state, 'state', 0);
        unset($state);

        whiteWaitStop();
    }
    unset($push);

    //loop
    motorUpStop();
}

//state 15
function whiteWaitStop()
{
    global $sleep;
    timerManage();

    //check if the white disk has been sorted
    if ($sleep == SORT) {
        //it has, so lets start moving the sorter down
        $temp = 10;
        storeData($temp, 'outputs', HBRIDGE1);
        $temp = 0;
        storeData($temp, 'outputs', FEEDERENGINE);
        unset($temp);

        //update state
        $state = 12;
        storeData($state, 'state', 0);
        unset($state);

        $sleep = 0;
        motorDownStop();
    }

    //loop
    $sleep++;
    whiteWaitStop();
}

//state 16
function motorDownStop()
{
    global $sleep;
    timerManage();

    //check if the sorter has moved down
    if ($sleep == TIMEMOTORDOWN) {
        //it has, so lets stop it
        $temp = 0;
        storeData($temp, 'outputs', HBRIDGE1);
        unset($temp);

        //update the state
        $state = 9;
        storeData($state, 'state', 0);
        unset($state);

        $sleep = 0;
        runningStop();
    }

    //loop
    $sleep++;
    motorDownStop();
}

//not a state
function timerInterrupt()
{
    timerManage();
    //show that we are in the timer interrupt
    $temp = 5;
    display($temp, 'display');

    //start moving the sorter up, to start the calibration
    $temp = 10;
    storeData($temp, 'outputs', HBRIDGE0);

    //stop the rest
    $temp = 0;
    storeData($temp, 'outputs', LENSLAMPPOSITION);
    storeData($temp, 'outputs', LENSLAMPSORTER);
    storeData($temp, 'outputs', LEDSTATEINDICATOR);
    storeData($temp, 'outputs', DISPLAY);
    storeData($temp, 'outputs', CONVEYORBELT);
    storeData($temp, 'outputs', FEEDERENGINE);


    //reset, because we will no longer be in timerInterrupt
    display($temp, 'display');
    unset($temp);

    //go back to initial
    $temp = getData('offset', 0);
    $temp2 = getFuncLocation('initial');
    $temp += $temp2;


    addStackPointer(2);
    pushStack($temp);
    addStackPointer(-1);
}

//not a state
function abort()
{
    //free some memory
    unset($engines);

    //prevent timerinterrupt
    setCountdown(1000);
    $temp = getData('stackPointer', 0);
    setStackPointer($temp);

    //stop everything
    $temp = 0;
    storeData($temp, 'outputs', HBRIDGE1);
    storeData($temp, 'outputs', HBRIDGE0);
    storeData($temp, 'outputs', LENSLAMPPOSITION);
    storeData($temp, 'outputs', LENSLAMPSORTER);
    storeData($temp, 'outputs', LEDSTATEINDICATOR);
    storeData($temp, 'outputs', DISPLAY);
    storeData($temp, 'outputs', CONVEYORBELT);
    storeData($temp, 'outputs', FEEDERENGINE);
    unset($temp);

    //apply the changes to actually stop it
    timerManage();

    //update the state
    $state = 17;
    storeData($state, 'state', 0);


    //show we aborted
    $state = 7;
    display($state, 'leds2', 0);
    unset($state);

    aborted();
}

//state 17
function aborted()
{
    //prevent timer interrupt
    setCountdown(1000);
    timerManage();
    //check if we can start again
    $startStop = getButtonPressed(0);
    if ($startStop == 1) {
        //start moving the sorter up, to start the calibration
        $temp = 10;
        storeData($temp, 'outputs', HBRIDGE0);
        unset($temp);

        //update the state
        $state = 0;
        storeData($state, 'state', 0);
        unset($state);

        initial();
    }
    unset($startStop);
    aborted();

}

//not a state
function timerManage()
{
    global $location, $counter, $engine, $sleep;

    if ($location == 0) {
        $engines = 0;
    }

    //makes sure that when $counter >12 it will reset to 0
    mod(12, $counter);

    //get the voltage of output $location
    $voltage = getData('outputs', $location);

    //power up the output when it needs to
    if ($voltage > $counter) {
        $voltage = $location;
        $voltage = pow(2, $voltage);
        $engines += $voltage;
    }

    //check if we did all outputs
    if ($location == 7) {
        //actually output the result
        sleep(1);
        display($engines, 'leds');


        unset($voltage);
        //check if abort is pressed
        $abort = getButtonPressed(1);
        if ($abort == 1) {
            abort();//STOP THE MACHINE!
        }
        unset($abort);

        //check if we are in a new iteration
        if ($counter == 6) {
            //set the first part of the display
            $temp = getData('state', 0);
            mod(10, $temp);
            display($temp, 'display', 1);
            unset($temp);
        }
        //check if we are at the end of the iteration
        if ($counter == 11) {
            //set the second part of the display;
            pushStack($sleep);

            $temp = getData('state', 0);
            //get the last digit of the state
            //we have no variables left, so we use $sleep

            $sleep = $temp;
            mod(10, $sleep);
            $temp -= $sleep;
            $temp /= 10;
            //display the last digit
            display($temp, 'display', 2);

            pullStack($sleep);
            unset($temp);
        }


        //set the variables for the next run
        $engines = 0;
        $location = 0;
        $counter++;

        //and return to where we came from
        return;
    }

    //loop
    $location++;
    branch('timerManage');
}
                // calibration
            storeData(1, "outputs", HBRIDGE0);
            //update the state
            $state = 0;
            initial();
        }
        //loop
        aborted();

    }

    void timerManage() {


        //make sure that when counter can not
        // be higher than 12
        mod(13, $counter);
        //get the voltage of output $location
        int $voltage = getData("outputs",
                               $location);
        //power up the output when it needs to
        if ($voltage > $counter) {
            $engines += pow(2, $voltage);
        }
        //check if we are in a new itteration
        if ($counter == 0) {
            //set the first part of the display
            $temp = getData("state", 0);
            mod(10, $temp);
            display($temp, "display", "1");


        }
        //check if we are at the end of the
        // itteration
        if ($counter == 12) {
            //set the second part of the display;
            $temp = getData("state", 0);
            $temp = $temp / 10;
            mod(10, $temp);
            display($temp, "display", "01");

        }
        //check if we did all outputs
        if ($location > 7) {
            display($engines, "leds", "");
            //set the variables for the next run
            $engines = 0;
            $location = 0;
            $counter++;

            //check if abort is pressed
            $abort = getButtonPressed(1);
            if ($abort == 1) {
                abort();//stop the machine
            }
            return;
        }


        $location++;
        timerManage();
    }
}
```