Building the Ultimate Machine using MATLAB/Simulink

Daan Meijers

June 8, 2018

Supervisor(s): Sebastian Altmeyer
Embedded system engineering is one of the largest fields in information technology. The impact of embedded systems can be found everywhere in the daily life; from household appliances like coffee machines to industrial machines such as car manufacturing robots. Therefore effective programming tools and skilled system engineers are key in the development of information technology integration in the human society.

Because embedded software is often complex model-based design is widely used to tackle this complexity problem. A widely known and powerful model-based design tool is Simulink. Simulink is a graphical programming environment developed by MathWorks. The tool has been proven to be very effective in automatic control systems creation, digital signal processing and model-based design. Real world examples of Simulink usage are the designing of spacecraft simulations, locomotives and factory plants[9, 1].

Unfortunately the opportunity to learn how to design embedded systems is not always present. For instance, at the University of Amsterdam is currently no embedded systems development course. In order to look at the potential value of creating the Ultimate Machine with Simulink, as an introductory project to embedded systems, this thesis has been written.

For this project the Ultimate Machine, as an example of a simple embedded system, has been programmed in order to look at the process of learning embedded systems programming, using model-based design. The hardware of choice was the Lego Mindstorms EV3, which is a programmable embedded platform. The Lego Mindstorms has been chosen because of the ease of building physical models and the presence of a support library in Simulink.

By building the Ultimate Machine a lot has been learned about both embedded systems development and model-based development. During the development of the machine many important topics came forward, such as dealing with real-world interactions, designing control logic, working with a model-based design tool and creating effective simulations. This project served as a good introduction to embedded systems engineering and model-based design.
Contents

1 Introduction 7

2 Theoretical background 9
   2.1 Embedded Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      2.1.1 Embedded Software . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
   2.2 Model-based development . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
      2.2.1 Principles of model-based development . . . . . . . . . . . . . . . . . . . . 10

3 The Ultimate Machine 11

4 Software and Hardware 13
   4.1 Lego Mindstorms EV3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
   4.2 MATLAB, Simulink and Stateflow . . . . . . . . . . . . . . . . . . . . . . . . . . 14

5 Development Process 15
   5.1 Work environment setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   5.2 Testing peripherals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   5.3 Physical Model - Ultimate Machine V1 . . . . . . . . . . . . . . . . . . . . . . . . 17
   5.4 Simulink Model - Ultimate Machine V1 . . . . . . . . . . . . . . . . . . . . . . . 18
      5.4.1 Main Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
      5.4.2 Subroutine Collection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      5.4.3 Modelling and Simulating . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

6 Results 29
   6.1 Physical Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   6.2 Simulink Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
      6.2.1 Press detector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
      6.2.2 Main Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

7 Reflection 33

8 Conclusion 35
CHAPTER 1

Introduction

With the increasing amount of embedded systems, there is a growing demand for embedded software. This trend is in turn paired with an increasing demand in embedded software engineers. However, since embedded software engineers are dealing with real-world constraints, developing such software is much more complex than programming for general-purpose computing. In order to reduce the level of complexity of software, it is becoming more and more common to use model-driven development techniques [5].

Because of these techniques and the dependency on the physical environment, the development of embedded software has become a distinctly different field of programming. This means that developers who are skilled in imperative or declarative programming are not necessarily also skilled in embedded software developing.

So with the current interest in embedded software developers it has become important for computer science students to at least touch the embedded software developing field during their study. Unfortunately this is currently not the case. The closest related course is currently the Electronics and Signal processing course taught at the Free University Amsterdam (VU).

Currently there are plans for an introductory embedded systems course for the Master Software Engineering at the University of Amsterdam. An important part of this course should be creating a basic embedded system. This thesis describes what comes into play when building such a system. Examples of important aspects of embedded software engineering are modeling, simulating, signal processing and control system designing.

In order to look at these parts an Ultimate Machine has been built using LEGO Mindstorms and Matlab/Simulink. The Ultimate Machine serves as a good introductory example as it is a very basic machine. The sole function of the Ultimate Machine is to turn itself off whenever it is turned on. This simple nature of the Ultimate Machine enables developers to design extensions to the machine to make it more interesting and sophisticated. The physical model has been built using the EV3, the third version of the LEGO Mindstorms programmable interface, together with LEGO Technic bricks. The software itself has been written using Simulink, which has proven itself as a professional modeling tool [1].

This thesis will start with providing background information about embedded systems and software, the Ultimate Machine, MATLAB and Simulink. Afterwards the development process of the Ultimate Machine will be described, where important aspects of embedded software development, model-based and Simulink programming will be explained in detail.
2.1 Embedded Systems

Embedded systems are computer systems which are designed to interact with the physical world. These systems are embedded into a larger system and are designed to perform a dedicated function. By integrating the hardware design, software design and control logic these systems are able to perform a dedicated function in the physical environment\[7\].

Embedded systems find their way in each aspect of our daily life. From kitchen appliances like ovens, induction cookers and coffee machines to automotive computers or industrial robots.

As technological advancements enable computers to become smaller, embedded systems are increasingly applied in the physical world. The embedded systems market is estimated to be worth 93.69 billion euros in 2017 and is currently growing by 4.05% annually. This growth is explained by the increasing use of embedded systems in the automotive industry, military, wearable market, smart appliances and health care equipment. Currently 98% of all microprocessors are produced for embedded systems\[5, 2\].

2.1.1 Embedded Software

The growth of the embedded system market is logically accompanied by an accelerating growth in the embedded software market. Cars for example, contain an increasing amount of micro controllers. In 2008, new cars contained on average 20-70 micro controllers, 100 million lines of code resulting in 1 gigabyte of software. Embedded systems and their software are currently the main drive for innovation in the automotive industry\[5\].

Because an embedded system usually interacts with the physical environment, it needs to take physical constraints in account. These physical constraints cause an increased amount of complexity. It is also often important to simulate the system before actually deploying it. Especially in fields like aerospace engineering it is not possible to test the software by deploying. It is also often difficult or impossible to update embedded software after deployment in this domain. Therefore, proper simulations are key in the development in embedded software. In order to tackle these problems model-based design is often employed to design complex embedded systems\[7\].
2.2 Model-based development

Software developers and researchers have always been abstracting program languages in order to shift the focus from program implementation to design. At first, program languages like Fortran were designed to shield the developer from having to program on a machine level. Nowadays program languages are further abstracted in order to increase the speed of development due to the decrease in complexity. Examples of abstracted programming languages are object-oriented languages like Java or C++. While these languages are successful in their task of elevating the abstraction level, further abstraction is still needed as their ability to mask complexity is exceeded by growing platform complexity. As a result developers are required to keep porting, adjusting and extending existing software. In order to do this, software developers have to work on a line-to-line basis, which costs valuable time they rather spend on the software design.

2.2.1 Principles of model-based development

An approach to address this challenge is model-based development. With model-based development the developer is able to focus on designing the software rather than focusing on the implementation. Model-based development enables this by combining the usage of domain-specific languages and automatic code generation. Domain-specific languages are computer languages which are focused on a particular domain, therefore often alleviating a part of the complexity of this domain. Without automatic code generation model-based development would be solely used to create documentation, like UML for example. While it is important and useful to have an accurately documented design, in reality the result usually differs from the intended design. Also does code generation lift the developers burden of worrying about implementation details.

There are a couple of important characteristics of model-based development. First of all it is important for the model to be an abstract simplified version of the system it represents. This is because model-based development is used to reduce the complexity of development. Furthermore it is also important for the model to be easily understandable. Especially when working in multidisciplinary teams, an understandable model aids the development process. This can be done by both ensuring the model looks simple and by providing ample documentation. While the model must be an abstraction of a system, it is still important that it is accurate. Models should be an accurate representation of a real world system. Other characteristics are that the model should be less expensive to construct and analyse, than its the real world version. It should also be verifiable through simulation, without having to deploy it first.
The Ultimate Machine, also known as the Useless Machine or Leave-Me-Alone Box, is a fairly basic machine with a low number of requirements. In its most basic form it will only have two interactions with the physical environment. The only input it requires from the environment is the detection of a button being pressed. In turn, the only reaction from the machine is the movement of a mechanical arm which pushes the button to return it to its original state. Because of these low requirements there is a lot of room for extensions on the machine, allowing developers to express their creativity. With each extension, the complexity of the machine can be increased, resulting in an increasingly sophisticated machine. Due to the simplicity of the Ultimate Machine computer science students can focus their work on the software design, rather than the design of the physical model. Even though in embedded systems the hardware and software are equally important, for computer science students the software side should be more interesting.

Another interesting aspect of the Ultimate Machine is its origin. The first useless machine as popularly known has been designed by Marvin Minsky in 1952[12]. He invented the machine while doing graduate work at Bell Labs, with his mentor Claude Shannon. Marvin Minsky is known as a pioneer in the field of artificial intelligence. He is, for instance, credited with the creation of the first neural network simulator (SNARC), the founding of MIT Computer Science and Artificial Intelligence Laboratory together with John McCarthy and has been awarded for his work with the Turing award[8]. Marvin Minskys mentor and computer science pioneer Claude E. Shannon was humored by Minksys Useless Machine and decided to build it. Shannon is known as one of the most influential scientists in the field of information science, therefore also called the father of information theory. Of his work the book The Mathematical Theory of Communication is the most notably as it serves as the foundation of information theory. He is also well known for his work on digital circuitry, where he proposed to use Boolean algebra in order to construct digital circuits.
The Useless Machine Claude Shannon built could be found on his desk, where many of his visitors could see this machine. Sir Arthur C. Clarke, a famous British science-fiction writer, most widely known for his book "2001: A Space Odyssey", also noticed the machine at Shannons desk. He wrote the following about what he witnessed:

I cannot leave Bell Labs without mentioning one more device which I saw there, and which haunts me as it haunts everyone else who has ever seen it in action. It is the Ultimate Machinethe End of the Line. Beyond it there is Nothing. It sits on Claude Shannons desk driving people mad. (Or sat, as Shannon is now at MIT.) Nothing could look simpler. It is merely a small wooden casket the size and shape of a cigar-box, with a single switch on one face.

When you throw the switch, there is an angry, purposeful buzzing. The lid slowly rises, and from beneath it emerges a hand. The hand reaches down, turns the switch off, and retreats into the box. With the finality of a closing coffin, the lid snaps shut, the buzzing ceases, and peace reigns once more. The psychological effect, if you do not know what to expect, is devastating. There is something unspeakably sinister about a machine that does nothing absolutely nothing except switch itself off.

- Arthur C. Clarke, The Ultimate Machine, August 1958
4.1 Lego Mindstorms EV3

The hardware of choice is the Lego Mindstorms EV3, the fourth generation of the Lego Mindstorms platform. The EV3 is a programmable interface also referred to as an intelligent brick. By using a set of sensors, motors and parts of the Lego Technic line the user is able to develop and build basic machines. The EV3 holds four input ports for sensors, four output ports for motors, contains a speaker and a small LCD display. It uses an ARM9 CPU which runs a Debian-based Linux operating system on default. An SD-card can be inserted to exchange files or boot another operating system. Users can connect to the EV3 using USB, Bluetooth and WiFi.

Lego Mindstorms has proven to be an useful tool for introductory embedded systems courses[6]. They are commonly used for simple projects to let people get familiar with robotics and embedded systems.

There are a couple of reasons why Lego Mindstorms is such an effective robotics learning tool, one of which being the familiarity of many users with Lego. Since a lot of people have played with Lego as child they are familiar with constructing objects from it. Therefore it is easier for most people to design the physical model of their system. A catch is however that Lego Mindstorms is mostly used with Lego Technic, which is overall less used and thus not always as easy as building something from Lego bricks.

Another advantage of the Lego Mindstorms is that it is possible to wirelessly communicate with the intelligent brick. As the brick is reachable using WiFi or Bluetooth the physical model has little impairment because of cables, which is sometimes critical for a device to function as intended.

One of the main reasons why Lego Mindstorms EV3 has been chosen for this project is due to the availability of MATLAB/Simulink integration. There is sophisticated support from programming tools for integration with the Lego Mindstorms devices. MATLAB/Simulink has for instance add-ons which provide the developer with function blocks for Mindstorms sensors and motors, for both the EV3 and its predecessor, the Lego Mindstorms NXT.
4.2 MATLAB, Simulink and Stateflow

The software for the Ultimate Machine has been developed using MATLAB, Simulink and Stateflow. MATLAB is a software environment for numerical computing, which is built around the MATLAB scripting language. It has been developed by MathWorks in the late 1970s and is considered a professional computing tool. MathWorks has developed a number of other tools, such as Simulink, which integrated in the MATLAB environment.

Simulink is a model-based design and simulation tool. It features a block diagram environment with a drag-and-drop interface. Simulink is used to model control software, signal processing algorithms, mechatronic systems and communication systems, either completely in Simulink or in combination with other MathWorks tools. System models are created using function-blocks, which originate from a customizable block library. Developers can create their own function-blocks, either by designing them in Simulink or coding them using MATLAB.

Simulink makes use of automatic code generation to generate C code from Simulink models. It also features a powerful simulation engine which can be used to verify the models at any step in the development process. Simulink offers tools for model-analysis in order to refine the architecture and increase the simulation speed. Simulink is able to simulate models using fixed-step and variable-step Ordinary Differential Equations (ODEs). ODEs are equations which describe dynamically changing phenomena, and often occur in fields like physics, astronomy, chemistry, biology and many more. There are multiple ways of calculating ODE values, and the efficiency of a solver depends on the equation itself.

Simulink has been proven to be a professional development tool. Various serious projects, like the designing of the European Space Agency’s ATV (Automated Transfer Vehicle) are praising Simulink for its usefulness. In the development process of the ATV, Simulink has been used in nearly all phases. It has been used for the specification, coding and design of the project. The C code which was automatically generated by Simulink ran within or faster than their hard real-time performance limits. The European Space Agency also credits Simulink for aiding with project management and teamwork[1].

The American multinational corporation General Motors has also been praising Simulink for its uses. By using MATLAB, Simulink and Stateflow they have supposedly shaved off twenty-four months of expected development time, resulting in a total development time of nine months. General Motors used these tools to design a control system architecture, model the control and diagnostics functions, and used the automatically generated production code created by MathWorks Embedded Coder[4].

Because of the proven professional applicability of Simulink it has been chosen as development tool for this project. It is interesting to look at the process of developing a simple machine like the Ultimate Machine using a proven sophisticated model-based development tool like Simulink. Another reason why Simulink is applicable for this project is the existence of a Lego Mindstorms integration add-on. MATLAB and Simulink both support the Lego Mindstorms NXT and EV3. This add-on supplies the developer with a set of function blocks which represent the various sensors and motors. It also provides special model templates which contain pre-set simulation parameters so Lego Mindstorms blocks can be simulated.

Another MathWorks product and intrinsic part of this project is Stateflow. Stateflow is an extension of Simulink which is used to design control logic by creating finite state machines and flowcharts. Stateflow enables the developer to create charts where these state machines and flowcharts can be designed in. These charts are placed in Simulink models where they can interact with other Simulink components.

While Stateflow looks simple at first its flexibility allows the creation of sophisticated charts. States can have sub-states, they can be placed parallel, and even overlap each other. Transitions between states are controlled by events, time-based conditions and external input signals. It is even possible to integrate MATLAB and Simulink functions within a Stateflow chart.
5.1 Work environment setup

Installing and setting up a Simulink project is fairly simple due to the installer software it provides. This installer installs all software for which the user has a license. For this project MATLAB version R2018a and Simulink version 9.1 were used. After installing the software, Simulink can be accessed by launching Matlab and opening the Simulink package in the MATLAB main menu. The Lego Mindstorms EV3 integration package could simply be downloaded using the add-on manager from MATLAB.

During the installation process no problems arose, however some issues came up when trying to connect Simulink with the EV3. Simulink should be able to connect with the EV3 using USB, Bluetooth or WiFi. Beforehand WiFi was chosen as the connection method of choice because of the advantage of not having to use cables. The Bluetooth adapter appeared to be failing, dropping the connection as soon as it was established.

Unfortunately connecting to the EV3 over WiFi was not as easy as it should be. Apparently the latest version of the EV3 firmware (1.09H) has some undefined behaviour with certain WiFi adapters, in this case a TP-Link adapter. While the EV3 could be pinged it could not establish a stable connection. The solution turned out to be downgrading the EV3 firmware to version 1.08H.

5.2 Testing peripherals

While developing the Ultimate Machine (or any embedded system), one of the most important steps is to figure out what values the peripherals generate or require. For instance, which can be expected from a touch-sensor, gyroscope or ultrasonic-sensor? Also, which values should be passed on to the motors and how do they react on each value? While the behaviour of these peripherals is described both in Simulink and the online Lego Mindstorms documentation, it is also important to test this behaviour out yourself. This way any inconsistent or undefined behaviour can be detected.
Motors

The documentation of the motors states that they can receive any value between -100 and 100, representing the percentage of power provided to the motor. All values between -100 and 0 will cause the motor to rotate backwards just as all values between 0 and 100 will cause it to rotate forwards. During the tests some interesting behaviour has been observed. For instance, any value between -10 and 10 will not cause the motor to move. Apparently such small power values are insufficient to make the motor turn. Also, the rotation speed depends not only on the power value provided, but on the current battery level of the EV3 itself. The input values simply indicate the percentage of power the EV3 has to dedicate to a motor. When the battery level is lower there is less power to provide, thus the motor will move slower despite the fact that the input values haven’t changed.

Touch-sensor

The touch-sensor is a basic push-button whose output value depends on whether the button is pressed or not. The output value is either 0 (off) or 1 (on). The most interesting traits which came forward during these tests were physical. Apparently there is a relatively strong spring present within the touch-sensor. Normally the spring in a pushing button does not exert a lot of opposing force. The touch-sensor does however, which could make it hard for a machine to keep it pressed. This problem came forward during the creation of the first physical model. This issue is emphasized by the fact that the button only triggers when it is pushed in deep.

Ultrasonic-sensor

The ultrasonic-sensor is used to measure distances by emitting sound waves and listening to their echos. It can detect an object at a maximum distance of 255cm. According to the Lego documentation it is also able to function as a sonar by emitting single sound waves and it can listen for other ultrasound waves. This is however where the limitations of the EV3 support-library comes into play. The support-library does not allow peripherals to switch modes. In the case of the ultrasonic-sensor it can only be programmed as a distance calculator. Another quirk of the ultrasonic-sensor is that when an object is too close it will output the maximum value. So when an object is placed against the sensor it will output a value of 255.

Gyroscope

The gyroscope has been the most interesting sensor to test. This sensor can be used to either display the current angle or the current rotation speed. The sensor is calibrated when the EV3 is booting or when it switches output modes. Unfortunately, just like with the ultrasonic-sensor is the gyroscope limited to one output mode in Simulink. In Simulink it can only be used to observe the current rotation speed. Another issue with the gyroscope is the inaccuracy of the sensor. The gyroscope emits a lot of noise while in rotation-speed mode. This problem came forward during the programming of the second model.
5.3 Physical Model - Ultimate Machine V1

The project started with the creation of a simple version of the Ultimate Machine. For this version the focus was on getting acquainted with Simulink. Therefore the physical model was designed as simple as possible, with only the most essential peripherals.

The hardest part in creating this version was the design of a toggle-system. There was no pre-built toggle-button, so this had to be built using other sensors. Eventually a system was designed which used two touch-sensors, one to detect the lever switching off and one to detect the lever switching on. This is where the counterforce from the touch-sensor comes into play. Because the touch-sensor has a strong spring the arm would need more power than desirable, thus it could not press the lever as gently as originally intended. Another problem was that the lever would be launched back by the spring, as soon as the user or the arm stopped pushing it. This was eventually partly countered by creating a bit of friction between the lever and the bottom. This caused the lever to get stuck in the middle, withholding it from being launched back completely. During the development of the Simulink model this second button has become obsolete however.

![Physical model of the Ultimate Machine V0 and V1](image)

Figure 5.1: Physical model of the Ultimate Machine V0 and V1
5.4 Simulink Model - Ultimate Machine V1

During the development process the software has had many different forms. There is a clear distinction between between the first and last models which were created, both in sophistication and clarity. This section will cover all the steps that were taking in the development process, from the first Stateflow chart to the final Simulink model of the first Ultimate Machine version.

![Simulink model of the Ultimate Machine V1](image)

The image above show the final model for the first version of the Ultimate Machine. The model uses two input parameters, the rotation of the motor and the touch-sensor. The output of the model is the power value for the motor. As can be seen from this model, only one button is used, while two buttons are present in the physical model. The reason for this will be explained later on in this section. This model has a few extensions built in, which enable the machine to use three different responses chosen at random. These extensions are implemented in the Subroutine Collection.

The model consists out of two blocks, the Main Control Stateflow chart and the Subroutine Collection subsystem. The following sections will explain how these blocks have been designed, what choices have been made and why. Also other information about the modelling and simulating process itself will be provided.
5.4.1 Main Control

The Main Control Stateflow chart has had many different forms over the course of the development process. Before the start of this project it was established that using Stateflow charts would be the most efficient way of controlling the Ultimate Machine. Therefore some experiments had to be run in order to figure out how these charts function.

Stateflow transition logic

One of the questions which came up was how to control state-transitions. Stateflow provides various methods for state-transitions; using conditional logic, event-based logic and temporal logic. These methods can be combined within a single chart, but there are limitations.

![Stateflow chart with event-based, conditional and temporal transition logic](image)

Figure 5.3: Stateflow chart with event-based, conditional and temporal transition logic

This Stateflow chart is the first attempt on designing the Ultimate Machine mechanism in Stateflow. The idea was that when a button is pressed, a **buttonPress**-event is launched, causing the chart to transition to the **START**-state. Then, if the **Counter**-value is higher than zero it will also automatically advance to the **FORWARD**-state. The **Counter**-value tells the system how long the motor (arm) will have to be running. This value is manually set during run-time in order to estimate the right value. The **after()** function is used to program temporal logic. This function will output false until after the specified time.

This chart did not work however. A Stateflow chart which uses events will only transition while an event is active. Therefore as soon as the **buttonPress**-event is disabled the chart stops executing, halting at the **FORWARD**-state. Also, since there are no transitions back to the **START**-state the chart can only execute once.

According to the Stateflow documentation it is wiser to use conditions to represent a change of input value, such as a button being pressed. Therefore a second Stateflow chart was created, which does work. When the chart is initialized it will follow the default transition to the **OFF**-state, where the motor speed is always zero. When the button is pushed the transition-condition holds true and the **REVERSE**-state is activated before transitioning back to the **OFF**-state after the same amount of time.

At first, using temporal logic to control the motors seemed like a valid strategy. However as stated before, the rotation speed of a motor is not only affected by the input value, but also by the current battery power. Therefore with varying battery levels the motor will rotate for different amounts in the same time.
Arm control mechanism

As stated in the previous section, controlling the arm by activating it for a specific time did not yield the desired results. In order to ensure that the arm would always go to the full length a second touch-sensor was placed. This touch-sensor would be activated when the arm pressed the lever far enough. While this did work as intended it was still not a perfect solution. Because of the spring in the touch-sensor the lever would be launched back as soon as the arm retracted. This has been partly solved by creating a bit of friction between the lever and the bottom of the Lego model.

Eventually the solution for this problem has been found in the Motor Encoder block. This Simulink function block, provided by the Lego Mindstorms add-on, outputs the current rotation of the motor. By specifying a minimum and maximum rotation and observing the current rotation the motor could be controlled precisely.

This chart is from the finishing routine of an old Stateflow model. When it receives the input value \( \text{run} = 1 \), it will start moving the arm forward by setting the power to 35. As soon as the current rotation is close to the maximum rotation the arm is put in reverse, until it comes close to the minimum rotation. Because there is always a minor delay between the software and the hardware the rotation limits have been widened.
In order to effectively estimate the maximum and minimum rotation for this model an initialization phase was developed. When the Simulink model is started the current rotation is set as the minimum rotation. Then the arm moves forward until the lever pushes the reset touch-sensor, setting the current rotation as the maximum rotation. While this could have been used to accurately estimate the rotation extremes, it still required the reset-button once. The solution for this problem also appeared to be an introduction into Simulink functions in Stateflow. The Stateflow chart below shows the final improved Initialization-phase, omitting the need for a reset button. This initialization phase does not only estimate the rotation extremes. It also estimates the least amount of power required for the arm to flip the lever. This is done to counter the impact of varying battery levels on the performance on the motor.

Figure 5.7: Initialization sequence from the Main Control chart in the V1-model.

The first two states within the Init-state are used to test the maximum rotation ranges. When this is done the following states are used to test the amount of power required. The keywords en and du are playing a major role here. Stateflow uses the keywords en (entry), du (during) and ex (exit) to indicate when lines have to be executed. Thus in the chart when the state RangeTestForward is activated the statement xmin = rotation is executed once. Then, while the state is active the statements power = 50 and diff = calcRotationDiff(rotation) are continuously called.

The function calcRotationDiff(rotation) is a Simulink function placed within the Init-state. This function receives the current rotation as input value and calculates the difference using the Signal Difference function block. The output result is the difference between each signal. Simulink functions in Stateflow are essentially triggered subsystems which can handle any input available within the current Stateflow scope. Because calcRotationDiff(rotation) is placed under the du keyword the diff-value is continuously updated. The diff-value is continuously evaluated by the transition condition [after(500, msec) & & diff == 0]. The after() function is used here to ensure that the arm has moved
before the condition is true. Without this extra evaluation the transition condition can evaluate as true before the arm has even moved.
Thus, when the rotation difference is equal to zero and at least 500 milliseconds have passed, the model can assume that the arm has reached the maximum rotation span.

The power test makes use of the same principle. This test starts the motor with a low power value, which will cause the arm to be pushed against the lever. However, when the arm is not moving and the current rotation is lower than the maximum rotation the model assumes that the lever is not yet switched, thus requiring more power. This causes the power value to be incremented until it is finally pushing the lever to the maximum rotation, indicating that the correct amount of power has been found.

By using this model various previously mentioned problems have been solved. There is now a guarantee that the arm will turn until it has pushed the button, which is a major upgrade from the time-based variant. There is also no longer a requirement for a reset-button, which simplifies the physical model. In addition to that the power test will ensure that the motor will always use sufficient power to turn the lever.

5.4.2 Subroutine Collection

The Subroutine Collection is another part which has seen multiple design iterations and tests before a final version was reached. The goal was to create a subsystem which would hold all the different responses by the machine. These responses would be conditionally executed, using a Simulink form of an if-statement.

![Diagram of Subroutine Collection]

Figure 5.8: The contents of the Subroutine Collection, where the different responses are selected by an If-block.
The Subroutine Collection currently holds three different responses, a basic response, delayed response and a motor sequence. When the active signal becomes positive an enabled subsystem starts outputting a value ranging from 1 to 3. An enabled subsystem will only activate upon receiving a positive control signal. The block will be active as long as this control signal is positive, as opposed to a triggered subsystem, which will only execute for a single time-step. Another difference is that while an enabled subsystem will not output values while its inactive, triggered subsystem will hold the last value since it was triggered.

The value which will be outputted from the enabled is processed by an If-block, which activates one of three If-Action blocks. The activated If-Action block will then execute, performing a response, before outputting a done signal. The done signals are merged before exiting the subsystem, ensuring that from the three If-Action blocks only one done signal is outputted.

If-block structure

As mentioned above, because the If-block results in multiple conditionally executed subsystems a Merge-block is necessary to merge the signals. Because of the nature of the Merge-block some design challenges arose however. There are some guidelines which the developer is required to follow in order to successfully use the Merge-block.

The most impactful guideline is that the If-Action blocks are required to hold their output when disabled. The done signal, originating from the If-Action blocks were designed as a notifier for the Main Control chart that the subroutine is done. Therefore this signal was not intended to be held, but rather deactivated once the Main Control chart is successfully notified. Because the done signal is held it would cause the Main Control chart to skip a subroutine the next time the Subroutine Collection subsystem is activated.

This problem could eventually be solved by introducing a reset signal. This signal would reset the output value of an If-Action block back to 0 before deactivating the If-block. The downside of this solution was that the Main Control chart needed an extra state, besides one to wait for the done signal to active, now also one to wait for the done signal to deactivate.

Feedback loops

The combination of the Main Control chart and the Subroutine Collection result in an algebraic loop. In Simulink an algebraic loop is a loop where all the parts have a direct feed-through. This means that the outputs of the Main Control chart and the Subroutine subsystem are dependent of each other. This dependency is not a problem in imperative languages, where the previous value is automatically used. But when an algebraic loop occurs in Simulink the model will be unable to compile.

This problem has been solved by placing a Unit Delay block in the loop. This block holds the done-signal value for one time-step, allowing the Main Control chart to evaluate the signal value of the previous time-step.

![Figure 5.9: The algebraic loop between the Subroutine Collection and the Main Control chart which has been solved by placing a Unit Delay.](image-url)
5.4.3 Modelling and Simulating

During the development of this model the power of Simulink as a modelling and simulation tool became apparent. The graphical environment makes it easy to distinguish function-blocks and the drag-and-drop interface enables quick and easy implementations. Blocks which share a common purpose can be grouped in highlighted areas or even in subsystems. Being able to test and validate a model without having to deploy it greatly reduced development time.

Model improvements

The final model shown at the beginning of this chapter was the result of a recreation of the original model, here referred to as the V0-model. The V0-model is the result of continuous extension of the default Lego Mindstorms Simulink template. One area of the original template is still present, which is the LEGO sensors area, here indicated in green. During the development of this model a new Stateflow chart has been created for each extension. In the process of extending this model it had transformed into an over-complicated model. Because of the interconnectivity of the different sections a lot of signal lines had to be drawn back and forth between each area. Due to the large amount of lines, the intersections of lines and the line density the model has become hard to read. In order to increase clarity the V0-model has been recreated with the V1-model as result.

![Figure 5.10: The original V0-model which had to be recreated.](image)
Model testing

The designed model can be tested in various way. During the development of the Ultimate Machine three methods of testing have been used. At first the model has been tested by simply deploying it to the hardware. Then, after discovering the usefulness of the external mode this feature has been used extensively. The model has also been designed to facilitate simulations without any hardware requirements.

Deploying code  At first all the models were tested by deploying them on the Ultimate Machine directly. Signal values of interest could be displayed on the EV3 display. This has proven very inefficient due to the long deployment time and the lack of flexibility of monitoring.

When the model is deployed it is first transformed into C code and then downloaded to the hardware. The automatic code generation process can consume a lot of time, depending on the model size. Any errors in the model are reported before this process, so the code generation process usually doesn’t fail. Downloading the code to the hardware can fail however, which happened occasionally. It often occurred that the EV3 was not reachable, while correctly connected via WiFi or USB. Usually the EV3 could eventually be reached after trying to deploy the model several times, but this problem made testing a time-costly process.

Another downside of this form of testing is the inflexibility of the monitoring signal values. Each time the developer wants to monitor a different signal the model has to be altered in order to display the desired value.

External mode  Simulink provides an external simulation mode, which can be used efficiently test a model on the hardware. External mode allows the user to monitor values and control flow in the Simulink interface while running the models on the hardware. The model can be deployed using any connection method but the resulting values have to be send using WiFi.

When simulating in external mode the developer is more flexible in monitoring values. Any signal can be selected in order to monitor its value. Stateflow charts can also be monitored in detail, showing the activate state. It is also possible to log a selection of signals, which can be viewed in real-time or after the simulation.

The downside of having to deploy the model on the hardware remains, so in some cases it would be preferable to fully simulate the model without any hardware intervention.
Full-virtual In order to simulate the Ultimate Machine V1 model without hardware it had to be altered slightly. The model requires two input values, a button press and the motor rotation, in order to know which step to take next. The button press can easily be simulated, Simulink provides a virtual button which can set a constant to a desired value. Virtualizing the motor rotation input value proved to be a bit harder.

![The Simulation setup for the V1-model](image)

The motor rotation is directly dependent on the motor power, thus a function has been built which calculate the new rotation value based on the current motor power. Because the simulation didnt necessarily needed to be precise in calculating the motor rotation the function has been kept simple. The new rotation is equal to the current rotation plus the current motor power. In order to incorporate the limits of the arm the rotation value is set to always be between 0 and 140. Because the motor needs a certain amount of power to push the lever a condition has been added, stating that the rotation can only exceed 100 if the motor power is 40 or higher. This function appeared to work which enabled the model to be tested without the usage of hardware.

One important feature to bear in mind while simulating is that Simulink will try to run the simulation as fast as possible, unless specified otherwise. This can be useful when a developer wants to run a simulation for a couple of time-steps and analyse the results afterwards. However, for this project it seemed more helpful to watch the simulation in real-time, so if the simulation gets stuck where this happens and under which circumstances.
Figure 5.12: The translation function which translates output values to valid input values.
CHAPTER 6

Results

After the original Ultimate Machine Simulink model was successfully recreated there was not much more work left to do on this model, besides creating more extensions. Therefore, in order to create a more sophisticated machine, a new Ultimate Machine was designed. This new version was designed to use as many peripherals as practicable. Also, since the Useless Machine is supposed to be a box, the model was designed in such a way that a box could be built around it. Another goal was to take the name leave-me-alone-box as literal as possible.

6.1 Physical Model

The Ultimate Machine V2 model is more complex than its predecessor, featuring a gyroscope, ultrasonic sensor, two medium motors and two large motors. Because of the aforementioned inconvenience of the touch-sensor it has been replaced by a gyroscope. The gyroscope is attached to the lever, so when it is flipped the gyroscopes rotation changes which can be interpreted as a button press. The mechanic arm has been extended with a latch, controlled by a second motor. This latch hides the arm when it is unused.

Figure 6.1: Physical model of the Ultimate Machine V2

The Ultimate Machine V2 model is more complex than its predecessor, featuring a gyroscope, ultrasonic sensor, two medium motors and two large motors. Because of the aforementioned inconvenience of the touch-sensor it has been replaced by a gyroscope. The gyroscope is attached to the lever, so when it is flipped the gyroscopes rotation changes which can be interpreted as a button press. The mechanic arm has been extended with a latch, controlled by a second motor. This latch hides the arm when it is unused.
The ultrasonic sensor is used to detect the hand of a user, which can be used to trigger an extra response. For instance, after the user has pressed the button several times the machine will drive away as soon as a hand is detected. This mechanic has been added to take the name leave-me-alone-box more literal, as the machine should now be able to actually run from the user.

In order to allow the machine to actually drive away a set of wheels has been added, controlled by two large motors. The large motors are important here as they are able to keep the wheels from turning when they are not powered, therefore ensuring that the user can press the button without pushing the machine.

6.2 Simulink Model

The V2-model could quickly be fabricated using the knowledge gained from developing the first few models. The result is a model which contains, despite the increased amount of features, less parts than the earlier models. Almost all functionalities could be incorporated in a single Stateflow chart.

Beside this one Stateflow chart there is only one other component, which is the button press detector. This detector observed the incoming values from the gyroscope and outputs a boolean signal whether a button press is detected or not.

6.2.1 Press detector

The press detector is used to determine whether the lever is flipped based on values from the gyroscope. From experiments with the gyroscope, it came forward that when flipping the lever the angle of the gyroscope changes on average by 15 degrees. Thus, the goal of this subsystem is to output a boolean signal true when a rotational difference of 15 or more is detected. Unfortunately several problems made this system more complicated than intended.

One of these problems is that the gyroscope is unable to output the current angle. Instead it is only able to output the current rotational speed. While the gyroscope itself is actually able to monitor both the Simulink EV3 support library only supports the detection of rotational speed.
In order to tackle this problem an integrator has been used. By integrating the rotational speed the current angle can be calculated. When the desired angle is reached the system outputs a true-value and resets the integrator. 

In theory this solution should work fine, unfortunately this is not the case in practice. The gyroscope is rather imprecise which, in combination with the integrator, causes a lot of false values. As an attempt on fixing this a simple noise filter has been created, which filters out all small values. This filter has increased the performance of the detector slightly, but it is still far from accurate.

![Diagram of the press detector which translates gyroscope data](image)

**Figure 6.2:** The press detector which translates gyroscope data

### 6.2.2 Main Control

The Main Control Stateflow chart now features three main states. The machine is either **Off**, **Running** (away) or **ButtonPressing**. Due to the new button pressing mechanism the arm motor will always need to be on full power in order to press it. The maximum rotation is also no longer necessary to determine. The arm will simply push the lever until no rotational changes can be detected. Since the arm will be moving on full power it is assumed that if the rotation doesn't change the arm has pushed the lever completely back.

The subroutine collection has been integrated in the **ButtonPressing**-state. Because of this there is no longer need for a separate subsystem, reducing the complexity of the model. The model features six different responses, of which five of them are special response-states. Each time when the **ButtonPressing**-state is activated a response is selected by random. First there is a chance that the response will be the standard response. This means that only the finishing sequence is activated, which is that the machine opens the latch, pushes the button and closes the latch again. If this response is not selected one of the other five responses is selected each with an equal chance.
The first three responses try to let the machine fool the user, by pretending to push the button. The first response tries to do this by opening the latch without actually doing anything. After a couple of seconds the latch will close again and it will wait a few seconds before starting the finishing sequence. The second response goes a bit further, it extends the arm until it is pressing the lever, but it will not fully push the lever. Then, the machine will retract the arm again and start the finishing sequence one second later. The third response will cause the latch to open for 30 degrees and lets it immediately close again. Then it will wait before starting the finishing sequence, just like the previous responses. The fourth response will hit the button repeatedly at full power and the fifth response will cause the machine to drive away before starting the finishing sequence.

When a subroutine is finished it will hook into the finishing routine, depending on how the subroutine leaves the machine. If the latch is closed the subroutine will start the finishing routine at the start, which opens the latch. If the latch is open at the end of a subroutine it will hook into the second state of the finishing routine. When this finishing routine is done the control will go back to the Off-state.

In the case that the button has been pushed several times the running state can be activated. When the ultrasonic sensor emits a value below 7 it is recognized as a hand and the machine will drive away until the hand can no longer be detected. It will do this twice before resuming to normal behaviour.

![Stateflow charts which handles the control logic in the V2-model](image_url)

Figure 6.3: Stateflow charts which handles the control logic in the V2-model
CHAPTER 7

Reflection

During this project a lot has been learned about embedded systems development and model-based development. A lot of knowledge has been gained on how to deal with real-world interactions, designing control logic, modelling and simulation.

One of the first issues which came forward during the project was that the motor did not always rotate with the same speed, despite the input value being constant. The cause for this appeared to be varying battery levels. This raised the need for calibration, resulting in the initialization phase of the V1-model, which also estimated the maximum possible rotation. While this solution worked well, it was eventually no longer needed due to the fact that the V2-model always required the motor to go full speed. Also, when the motor is going full speed and the rotation difference is zero, the arm is assumed to have reached its maximum rotation. Therefore the maximum rotation estimation also turned obsolete. Despite the fact that the V2-model no longer needed calibration, this method is still considered useful in the case of the V1-model.

A recurring problem during the design of the Ultimate Machine was designing a lever. At first a touch-sensor was used to detect the lever being switched, but this sensor had two major drawbacks. The sensor was hard to activate, it had to be pressed in deeply in order for it to trigger. At the same time it would launch back the lever as soon as it was let go because of the strong spring inside. This problem was eventually solved by using a gyroscope instead of a touch-sensor. The gyroscope outputs the current rotation speed, but the model required a boolean signal to indicate the lever being switched. In order to translate the input data from the gyroscope, a translation function was built. Because of this solution the lever is no longer launched back. A drawback which has come forward instead is that the machine sporadically activates without being actually activated, because of the inaccuracy of the gyroscope.

Since conditional statements are major features of programming languages, it seemed interesting for this project to look at how If-statements work in Simulink. Since the subroutines of the Ultimate Machine have to be run conditionally, an If-block has been used here. Unfortunately the If-block did not work exactly as expected and caused the system to become unnecessarily complex. The need to hold a signal in order for the corresponding Merge-block to work, raised the need for a reset value. Therefore, in order to reduce the complexity of the model, the collection of subroutines has been placed in Stateflow. The result is a less complex model which is much easier to comprehend.

Designing Stateflow charts proved to be the most important part of creating the Ultimate Machine. Over the course of the project, a lot has been learned about effectively creating simple yet sophisticated Stateflow charts. The importance of carefully designing decision logic became quickly apparent. For instance, it is
important to keep in mind that when evaluating input values, conditional logic should be used rather than event-based logic.

The en, du and ex keywords appeared to be very useful, especially when trying to keep Stateflow charts compact. For instance, in order to monitor the arm or latch rotation in the V2-model, the function \texttt{calcRotationDiff()} is repeatedly called by placing it under the du keyword. Likewise, by setting a value on entry and unsetting it on exit, a value can be set temporary within a single state. Without these keywords an extra state would be needed to unset the value.

The use of Simulink functions within Stateflow also proved to be very helpful. For instance, in order to calculate the rotation difference the Simulink function \texttt{calcRotationDiff()} is called. While this function could also have been placed outside of Statelflow, it would cause the Stateflow chart to need an extra input port, thus making the model unnecessarily more complex.

It also proved to be more effective to use a single bigger Stateflow chart than multiple small ones. For instance, the V0-model consisted out of four different Stateflow charts. The idea behind this was to create a different Stateflow chart for each functional part of the model, making it easy to see which chart does what. This caused the model to become extremely hard to read, as the amount of signal lines made it very messy. From the first redesigning phase came forward that creating a single large Stateflow chart, as opposed to multiple smaller charts, increases the quality of the model.

The fact that Simulink allows extensive personal documentation within the model is really useful. Blocks can be grouped in color coded areas, whole paragraphs can be written between blocks and the font sizes from nearly all visible text can be altered. These features help making sense of the model when it is getting complex, which is an important feature within a model-based environment.

However, one cannot rely too much on this, as can be seen in the V0-model. Even though all the different parts had been placed in colored areas with descriptive names the model still got too complex. Therefore establishing hierarchy can be just as important, if not more, as documenting. The impact of establishing hierarchy can best be seen when comparing the V0 and the V1 model. The V1-model looks extremely simple as opposed to the V0-model. Creating hierarchy has also proven to be useful on a functional level. By creating a subsystem from the V1-model it became possible to create a full-virtual simulation setup. Making a full-virtual setup no longer meant having to alter model itself, but rather switch a couple of input and output blocks.

Being able to simulate the model, either fully virtual or using external mode has proven to be extremely helpful in the development process. Both modes have been used interchangeably as they serve different purposes. First, in order to check whether the model works it was most useful to test the model only virtual. Then, if the model seems to work it could be tested on the hardware using external mode to see if the right values have been used. When simulating fully virtual it is important to be able to generate valid input values. In order to this for the Ultimate Machine the power output has been translated to the rotation input, since these are closely related. The translation function has been kept simple because there was no need for a lot of precision.

Two downsides of Simulink which came forward quite often were the product-based approach of MathWorks and the lack of informational error messages. It has often occurred that while trying to deploy the model to the hardware, Simulink prompted that it required Simulink Coder, which was not the case. Simulink Coder is a package which was not included in the license provided by the university, thus this package had to be bought. Eventually Simulink Coder didn’t even appear to be needed. Likewise, Simulink once prompted to restructure the model, which would result in a model which couldn’t run. The reason it wouldn’t run was that it was now using a certain data-type which cannot be used unless another product was installed. There was also often a lack of proper error messages. Any error within a Stateflow chart would result in a message that the Stateflow chart didn’t work, without any specification of what and where. At the same time sometimes the error messages became even as vague as: “***Model Error***”.

34
The goal of this project was to create the Ultimate Machine using Simulink, which has been successfully accomplished. Simulink is an established professional modelling and simulation tool, therefore it was interesting for this project to see how a Simulink novice becomes able to build a machine.

The focus of this project was on the software side of embedded system development. This was emphasized by the choice of hardware. Lego Mindstorms is easy to build with, both because of the nature of Lego and the experience of playing with Lego during childhood. It was however still important to design this system as a tightly coupled combination of hardware and software. Physical constraints and hardware limitations had to be accounted for during the development of Simulink models. Both the issue with the gyroscope and the use of the initialization phase in the V1-model are examples of this.

In order to create the Ultimate Machine a basic understanding of Simulink had to be established. There was no prior knowledge of Simulink, so all Simulink programming knowledge has been obtained during the building process of the Ultimate Machine. Therefore the process of learning Simulink is well portrayed in this paper, where the quality of the V0, V1 and V2 models are a measurement of the current level of Simulink experience.

Stateflow appeared to be the most important tool for creating Simulink models. The use of state machines was key in controlling the Ultimate Machine. While Stateflow appears simple it is powerful due to its ability to create complex structures. Through Simulink and MATLAB integration there is hardly anything which cannot be built in Stateflow. For instance, most structures built in Simulink could eventually be substituted by Stateflow charts. It is probably even possible to move the Press-Detector from the V2-model to the Main Control Stateflow chart, although this would not necessarily be beneficial for the model.

Making sure the models are clear and understandable is key to model-based design. This could be easily established through creating hierarchy and proper documentation. Simulink allows the developer to make the models as comprehensible as possible through subsystems, color coded areas and text fields.

Simulating models was also extremely useful in Simulink. The use of multiple types of simulations help testing different elements of a model. Unfortunately the Simulink error messages are really poor. This makes it hard to pinpoint faulty subsystems, thus it is often helpful to test all subsystems separately before joining them.
To conclude this thesis it can be said that building a simple machine such as the Ultimate Machine using a professional tool as Simulink is an extremely informative experience. Plenty of insight into embedded software development has been gained, in addition to a bit of embedded system engineering in general.

This project is well suited to be used as an introduction to embedded software development for students. The use of Lego Mindstorms eases the creation of physical models, so they can be built quickly and be easily modified during development. The basics of developing with Simulink are not hard to learn, students should be able to figure out a lot about Simulink by just tinkering with it. Stateflow is virtually indispensable for this project. It would be useful to know some details about Simulink and Stateflow beforehand however. A basic understanding about feedback loops, If-blocks, subsystems, and basic modelling and simulation principles could help students to get started. Also certain features of Stateflow, like the various forms of transition logic, the use of Simulink functions and the en-, du- and ex-keywords are useful to know in advance. This thesis can be used to provide the necessary information.


