FH Joanneum

Graz

Model Based Design

Numeric Sequence Lock

Training Unit 04

Authors
David B. Heer
Jakob Soukup
Graz, January 9, 2019

Lecturer
Alfred Steinhuber

Contents

Introduction

The task was to implement a Finit State Machine in Simulink. Afterwards this Finit State Machine is to be tested for its requirements. Finally code should be generated and loaded onto the Arduino board.

1 Requirements

In the next Subsections 1.1 & 1.2 the requirements are explained in detail and sorted. In the last Subsection 1.3 is a list with missing requirements.

1.1 Order of Priorities

In this subsection the requirements are sorted according to their priorities.

1. **Unlocking**
   This is the core of the task and so the most important requirement.

2. **Locking**
   It is necessary to return to the locked state.

3. **Sampling**
   With a sampletime of 10ms you do not have to debounce the inputs. In this case is sampling important.

4. **Wrong Sequence**
   This requirement makes it difficult to crack the code using the brute force method.

5. **Input Handling**
   This is a nice feature and can be useful if you have mistyped.

6. **State**
   It would be a nice feature to see the current state. But it is not essential.

7. **Keypad**
   It is absolutely irrelevant whether the numbers are compared as integers between 0-9 or as ASCII integer numbers between 48 - 57.

The Voltage Monitoring is an own requirement. Because it has no connection with the remaining requirements.

1.2 Funcional and non functional requirements

The requirements from the Word-document where splited in functional and non functional requirements.

1.2.1 Functional

Unlocking Configurable sequence and the change of the output is functional

Wrong Sequence To blocking the entry for 5s after three wrong sequence entry is functional

State Make the current state visible is a functional demand

Input Handling To detect input changes is functional

Voltage Monitoring To show the voltage from the power supply is a functional requirement

1.2.2 Non functional

Unlocking It is non functional to enter the sequence by the button #
Should the output be true or false after the correct sequence is non functional

State To make the state by an output visible is non functional, you could use an LED on the board or so

Sampling It is a resource requirement

Locking It is not functional with which button the lock can be closed

Keypad It is an additional requirements, an other possibility whould be a wheel or a fingerprint scanner
It is also not a system requirement whether the data comes in ascii or binary or whatever

Voltage Monitoring A battery as power source is not a functional system requirement
A functional system requirement whould be a portable power source

1.3 Missing requirements

Some missing requirements are listed here:

- Minimum an maximum voltage
- To change the sequence while running the system
- The power source have to be portable or not
- Alarmsystem in case of wandals
- Size of sequence
- Area of use

2 Stateflow implementation

Based on the previously defined requirements the state machine of the numeric sequence lock shall be implemented in Simulink using stateflow. This task begins with defining the needed cases and then working on the transitions between said cases.

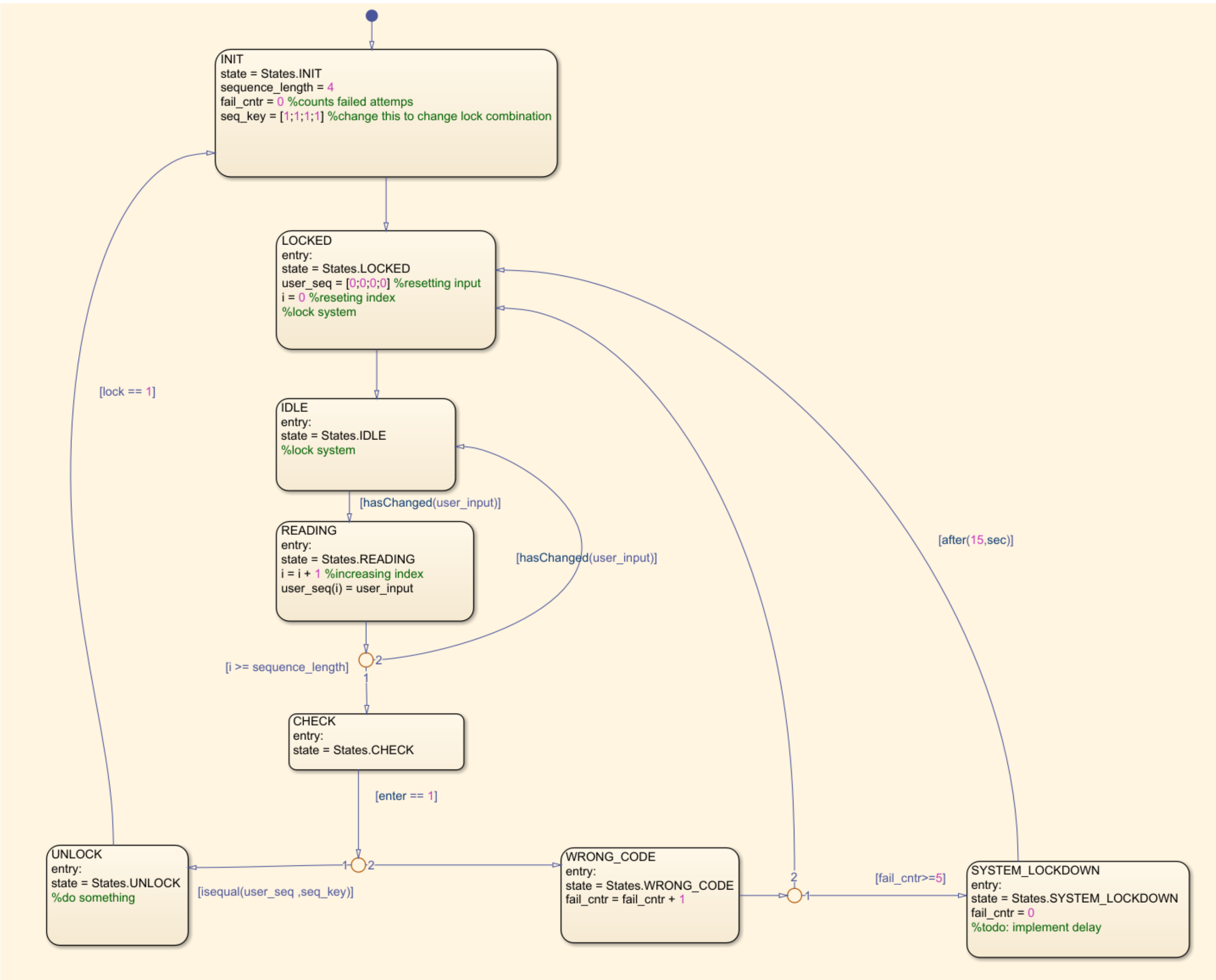

Figure 1: *stateflow implementation of the state machine*

2.1 Statemachine detailed

In order to further understand the working of the models, each case will be briefly explained and reviewed. An overview is in Figure 1.

<u>INIT</u>

As the name implies, this state sets the initial condition for the lock. The fail counter is reset to zero and other initial settings such as the sequence length and the sequence key are set in here. The Transition to the next state is made immediately after setting all parameters, without any conditions.

LOCKED
The system gets locked and the buffer for the user input is reset. The index i is used to keep track of how many digits were already entered. Again, the transition to the next state is done immediately after finishing all tasks without any conditions.

IDLE
This state is used to wait for user inputs, other than that it serves no purpose. After a user input was detected, the transition to the READING state is made.

READING
Here the detected input is stored in the previously declared buffer and the index is increased. If the index exceeds the sequence length, the state machine transitions to the CHECK state, if not, and the user input changes again (button was released) it moves back to the IDLE state and waits for the next digit.

CHECK
In this state the machine waits for the user to activate the enter key to confirm the input. After receiving it the sequence is compared to the key and if it matches the machine moves to the UNLOCK state. If the sequence is wrong, it moves on to the WRONG_CODE state.

UNLOCK
This is the state in which, if there was an actual locking hardware attached, the system unlocks itself. After completing the unlocking phase, it moves back to the INIT state to reset the system.

WRONG_CODE
Here the fail counter is increased and if it exceeds the allowed fail attempts, it moves to the LOCKDOWN state. If the limit Is not yet reached the machine jumps back into the LOCKED state in order to allow another try.

LOCKDOWN
The system waits for 15 seconds before returning into the LOCKED state. In addition, the fail counter is reset.

2.2 Input translation

Since the Arduino later will receive ascii values for the available some sort if translation to the system must be implemented in order to feed the statemachine the correct inputs. This was done by creating a simple subsystem in Simulink, evident in Figure 2.

The different inputs are filtered by their ascii value and fed into their outputs. The star an the hashtag are rather obvious. The interval filters for the ascii value of all 10 digits and the resets them to values from zero to nine.

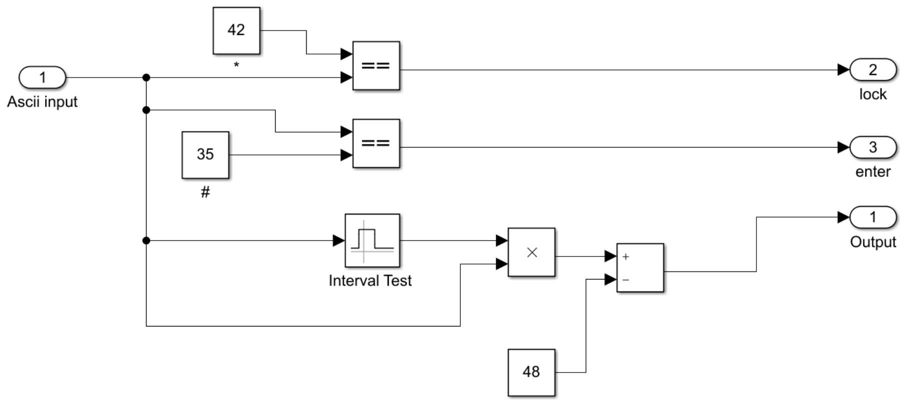

Figure 2: *check input subsystem in Simulink*

2.3 Testing

In order to test the system without its hardware a constant was attached to the input of the check input subsystem. This system is shown in Figure 3. The constant got linked to multiple push buttons which made testing the functionality possible.

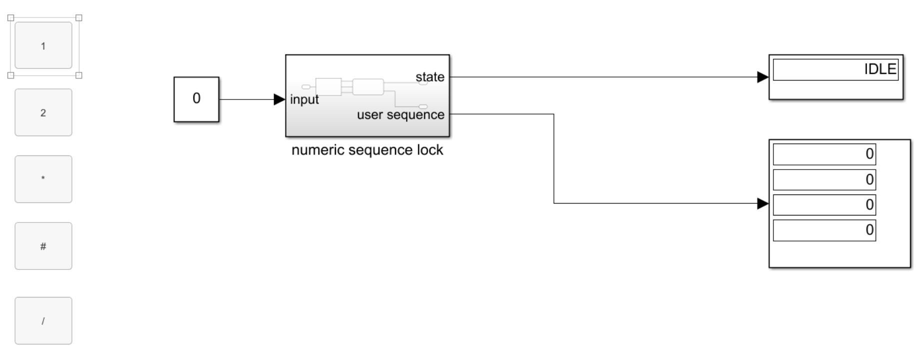

Figure 3: *Test structure of the sequence lock*

All possible cases, shown in Figure 4, where tested and the system held up to the expectations. If the correct sequence was entered and the hashtag was pressed the system unlocked itself. The input of the '/' character was ignored. After then pressing the start button the system was locked again. If the wrong sequence was entered the system returned into the IDLE state. If the wrong code was entered too many times the LOCKDOWN state was entered. However, the time the system spent in this state was way too short due to Simulink not simulating in real time.

Stage0	Stage1	Stage2	Stage3	Stage4	SeqFound Flag
0	3	2	1	3	True
X	3	2	1	3	False
0	X	2	1	3	False
0	3	X	1	3	False
0	3	2	X	3	False
0	3	2	1	X	False

Figure 4: *Test Cases*
Source: Numeric Sequence Lock Task Skript

2.4 Arduino implementation

After having a working model of the sequence lock the next step was to transfer the software onto the actual hardware, this time being an Arduino UNO with a numerical pad attached. Simulink shall automatically generate the required C code. For this purpose the number pad was attached to the Arduino as in Figure 5.

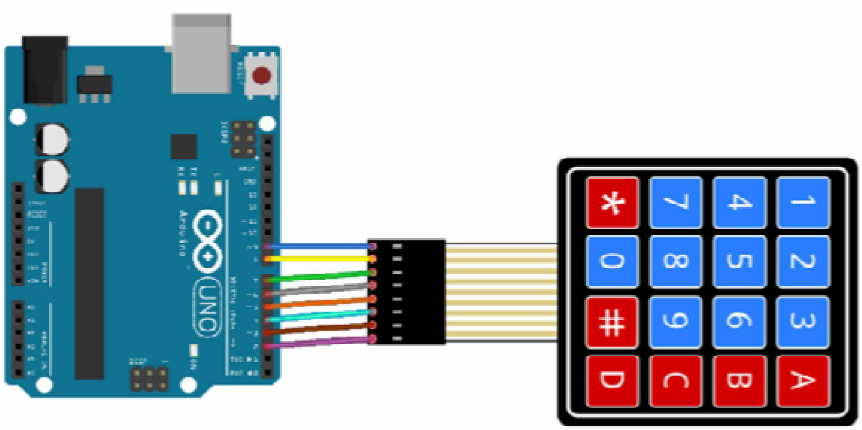

Figure 5: *pin assignment*
Source: ECE MBD: Laboratory session

The given template for the keypad was included into the Simulink model and the settings were adjusted in order to motivate Simulink to deploy the code onto the Arduino. Visible in Figure 6 and 7.

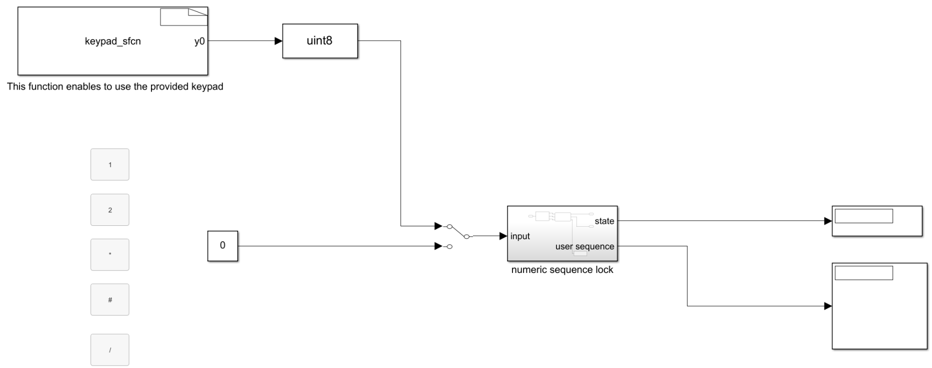

Figure 6: *Implementation in simulink for arduino*

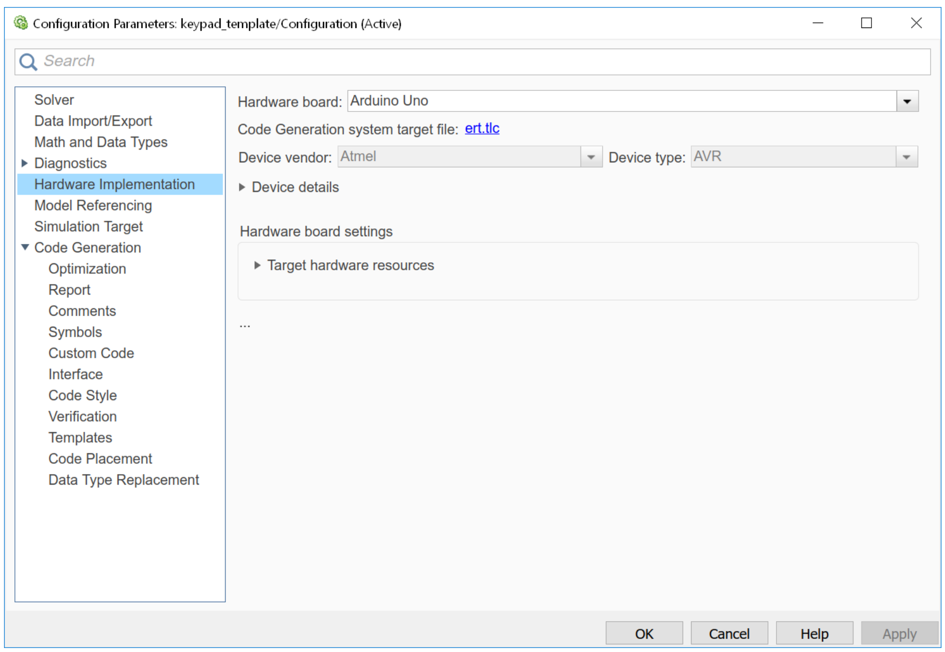

Figure 7: *simulink settings*

In addition, the compile mode has to be set to external. After that the model was deployed on the Arduino and was ready for testing. In order to check the correct working behavior, the same procedure as before was used. Only this time the buttons on the keypad where pressed as opposed to the ones in the Simulink model.

3 Voltage Monitoring

Due to the simplicity of the voltage monitoring requirements, no stateflow was implemented to do this. For this example were seted 3.3V as minimum and as maximum 3.7V. In Figure 8 is the

implementation evident. The voltage input is compared with the minimum and maximum value. If the input value ist higher or lower then then a rectangular signal is emitted on the respective output. This signal is boolean and triggers all 0.5sec.
Slight voltage fluctuations were neglected. Therefore no schmitt-trigger was realized. Maybe it could be that the output sometimes flashes only very quickly or a short time.

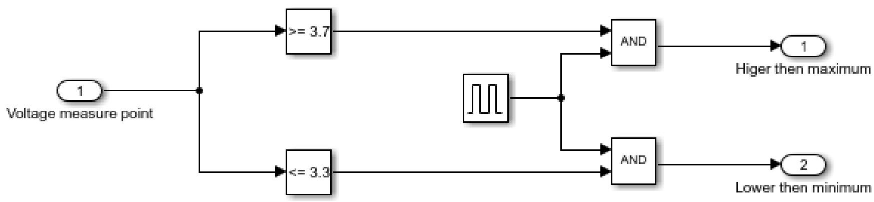

Figure 8: *Implementation of the voltage monitoring*
Source: Own presentation

3.1 Testing the Voltage Monitor

In Figure 9 is a test setup for the voltage monitor. As input a sinus generator was used. The amplitude was 1.5V with an offset was 3.5V, so the middle between minimum and maximum. The frequence is $1^{rad}/_{sec}$. The results of this test is visible in Figure 10.

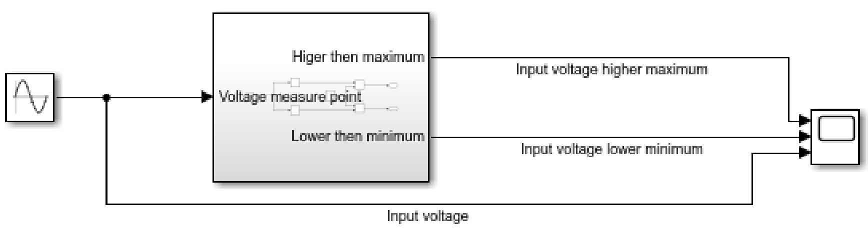

Figure 9: *Test structure*
Source: Own presentation

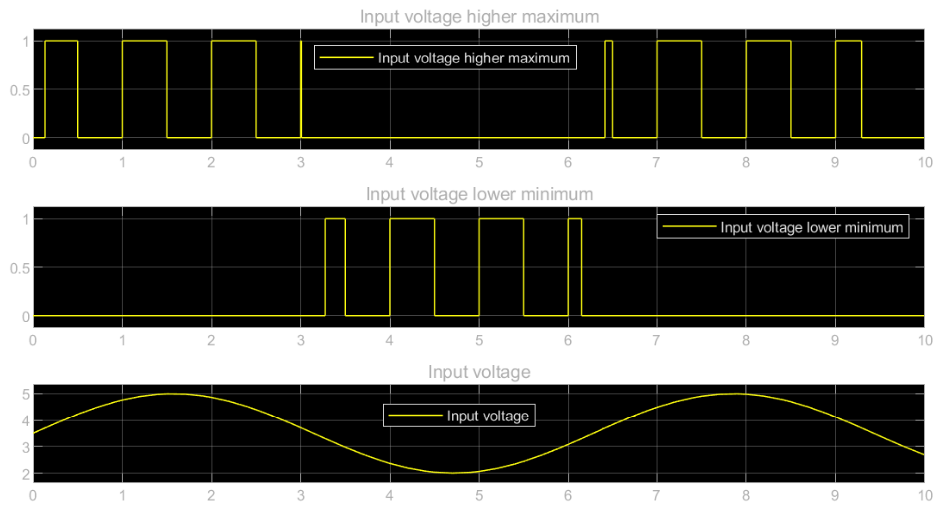

Figure 10: *Plot of the in- and outputs. The two upper plots are boolean depence of time. The third is voltagein dependence of time*
Source: Own presentation

4 Conclusion

Executable program code was generated in a few steps. In this case a statemachine as control was very helpful and easy to implement.

1. Requirements were created and sorted
2. A model was built and tested and verificated
3. Executable programcode was automatically built
4. The program code was loaded to a microcontroller and validated

Thanks to the simulation, the program code on the device worked right away. With a few adjustments, the code could be loaded onto other devices. Additional an indipendently voltage monitor was created with standart library blocks.

List of Figures