**Dokumentation Reaktionsspiel -**

**Konzeption der Zustandsmaschine(n)**

**Projektarbeit**

im Studiengang  
IT-Sicherheit und Mobile Systeme

vorgelegt von

**Johannes Schumacher**Matr.-Nr.: 16452

am 19. Februar 2020   
an der Hochschule Stralsund

Professor: Prof. Dr.-Ing Uwe Creutzburg

# Ehrenwörtliche Erklärung

„Hiermit versichere ich, Johannes Schumacher, ehrenwörtlich, dass ich die vorliegende Projektarbeit mit dem Titel: „Konzeption der Zustandsmaschine(n)“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Stralsund, den 19.02.2020

# Kurzfassung

Folgendes vorgestelltes Dokument beinhaltet die Projektarbeit des Moduls Embedded Systems im Studiengang IT-Sicherheit und Mobile System an der Hochschule Stralsund. Die Arbeit ist ein Lösungsvorschlag zur Entwicklung einer Zustandsmaschine als Teil eines Reaktionsspiels. Der Erstellung dieses Dokuments voraus ging die Erstellung eines Pflichtenhefts und ein Anforderungsheft. Die Lösung der Arbeit ist implementierbar auf einem Mikrocontroller MSP430 von Texas Instruments. In dieser Dokumentation wird zunächst eine Zustandsmaschine, basierend auf den Lösungsvorschlägen im Pflichtenheft, geplant. Anschließend wird der Quellcode mit einem Code Generation Tool erstellt und zuletzt wird diese erstellte Zustandsmaschine auf dem MSP430 implementiert und getestet. Neben der eigentlichen Erstellung der Zustandsmaschine beinhaltet diese Arbeit Gedanken zu dem gesamten Ablauf des Reaktionsspiels auf Grundlage der Featureliste im Anforderungsheft. Alle weiteren Softwaremodule können auf Basis dieser Zustandsmaschine erstellt werden. Auf diese Weise wird eine Modularität erzeugt, die das Schreiben einer lesbaren, anpassbaren und erweiterbaren Software ermöglicht und somit die Qualität der Software erhöht bzw. anschließende Implementierungen sich, aufgrund geringer Abhängigkeiten, vereinfachen.

**Schlagwörter**: Zustandsmaschine, Eingebettete Systeme, Zustandsmaschinengenerator, Codegenerator, Generator, Mikrocontroller, MSP430

# Abstract

Subject of the following theses is the design of a Final State Machine. This includes the aspects of an initial planning, the usage of a generator of code for the Final State Machine and an actual implementation of the Final State Machine.

**Keywords:** Final State Machine, Embedded Systems, FSM Code Generator, Microcontroller, MSP430

# Inhaltsverzeichnis

[Ehrenwörtliche Erklärung 2](#__RefHeading___Toc8485_1087498754)

[Kurzfassung 3](#__RefHeading___Toc8487_1087498754)

[Abstract 3](#__RefHeading___Toc8489_1087498754)

[Inhaltsverzeichnis 4](#__RefHeading___Toc8491_1087498754)

[Abkürzungsverzeichnis 7](#__RefHeading___Toc8495_1087498754)

[1 Einleitung 8](#__RefHeading___Toc8497_1087498754)

[1.1 Einführung 8](#__RefHeading___Toc250_1822160454)

[1.2 Ziel 8](#__RefHeading___Toc252_1822160454)

[1.3 Abgrenzung 8](#__RefHeading___Toc254_1822160454)

[1.4 Pflichtenheft 8](#__RefHeading___Toc614_1054882913)

[2 Dokumentation 9](#__RefHeading___Toc1432_2357535209)

[3 Entwurf 10](#__RefHeading___Toc256_1822160454)

[3.1 Entwicklungsumgebung und Programmiersprache 10](#__RefHeading___Toc258_1822160454)

[3.2 Der MSP430 10](#__RefHeading___Toc260_1822160454)

[3.3 Zustandsmaschine 10](#__RefHeading___Toc262_1822160454)

[3.4 FSM Generator 10](#__RefHeading___Toc264_1822160454)

[3.5 Benutzerinteraktion 11](#__RefHeading___Toc616_1054882913)

[3.6 Versionierung 11](#__RefHeading___Toc322_1822160454)

[3.7 Quelltextdokumentation 11](#__RefHeading___Toc618_1054882913)

[3.8 Diagramme 11](#__RefHeading___Toc620_1054882913)

[3.9 Testfälle 11](#__RefHeading___Toc622_1054882913)

[4 Durchführung 13](#__RefHeading___Toc266_1822160454)

[4.1 Ausgangslage 13](#__RefHeading___Toc268_1822160454)

[4.2 Die Zustandsmaschine im Überblick 13](#__RefHeading___Toc324_1822160454)

[4.2.1 Erstellung 13](#__RefHeading___Toc337_1822160454)

[4.2.2 Die Zustände 14](#__RefHeading___Toc624_1054882913)

[4.2.3 Die Events 15](#__RefHeading___Toc626_1054882913)

[4.2.4 Die Zustandsübergangsfunktionen 15](#__RefHeading___Toc628_1054882913)

[4.3 FSM Quellcodegenerator 16](#__RefHeading___Toc341_1822160454)

[4.4 Implementierung mit Code Composer Studio 17](#__RefHeading___Toc630_1054882913)

[4.4.1 Projekteinrichtung 17](#__RefHeading___Toc632_1054882913)

[4.4.2 Event-Aufrufe in der main.c 17](#__RefHeading___Toc634_1054882913)

[4.4.3 Nachvollziehbarkeit der Events 17](#__RefHeading___Toc636_1054882913)

[4.4.4 Generierung der Quellcodedokumentation 17](#__RefHeading___Toc638_1054882913)

[5 Funktionales Testen 18](#__RefHeading___Toc1434_2357535209)

[6 Auswertung 19](#__RefHeading___Toc1436_2357535209)

[6.1 Erreichen des Projektziels 19](#__RefHeading___Toc1438_2357535209)

[6.2 Folgeentwicklungen 19](#__RefHeading___Toc1440_2357535209)

[6.2.1 Entwicklung der Dialogstruktur mit UART 19](#__RefHeading___Toc2294_2357535209)

[6.2.2 Parametersatz 19](#__RefHeading___Toc2296_2357535209)

[6.2.3 Implementierung des Spiels 19](#__RefHeading___Toc2298_2357535209)

[6.2.4 Ausnahmebehandlungen 20](#__RefHeading___Toc2300_2357535209)

[6.3 Erkenntnisse 20](#__RefHeading___Toc1442_2357535209)

[Anhang A: Pflichtenheft 21](#__RefHeading___Toc8499_1087498754)

[Anhang B: FSM-Excel-Schema 28](#__RefHeading___Toc8501_1087498754)

[Anhang C: Generierter Quellcode der FSM 29](#__RefHeading___Toc13635_1087498754)

[C.1 fsm.h 29](#__RefHeading___Toc8505_10874987541)

[C.2 fsm.c 31](#__RefHeading___Toc8507_10874987541)

[C.3 fsm\_transition.c 34](#__RefHeading___Toc8509_10874987541)

[Anhang D: Testen des Zustands und der Übergänge 36](#__RefHeading___Toc8511_1087498754)

[D.1 ActState 36](#__RefHeading___Toc8513_1087498754)

[D.2 Zeiger auf Funktionsaufruf 37](#__RefHeading___Toc8515_1087498754)

[Quellenverzeichnis 38](#__RefHeading___Toc8517_1087498754)

Abbildungsverzeichnis

[Abbildung 1: Dialogstruktur 11](#Abbildung!0|sequence)

[Abbildung 2: Zustandsdiagramm 13](#Abbildung!1|sequence)

[Abbildung 3: Eventbus für Android 20](#Abbildung!2|sequence)

Tabellenverzeichnis

[Tabelle 1: Zustände 14](#Tabelle!0|sequence)

[Tabelle 2: Events 15](#Tabelle!1|sequence)

[Tabelle 3: Transitionsaufrufe 16](#Tabelle!3|sequence)

# Abkürzungsverzeichnis

FSM Final State Machine

CCS Code Composer Studio

RAM Random Access Memory

UART Universal Asynchronous Receiver Transmitter

HTML Hypertext Markup Language

CRC Cyclic Redundancy Check

ROS Robot Operating System

LED Leuchtdiode

# Einleitung

## Einführung

Im Rahmen der Veranstaltung Embedded Systems SMS6300, beschreibt folgende Dokumentation die Umsetzung des Reaktionsspiels mit dem MSP430.

## Ziel

Thema der Arbeit ist die Konzeption einer Zustandsmaschine. Ausgehend von einer initialen Planung der Zustandsmaschine wird der Quellcode mit dem FSM Generator von Texas Instruments erstellt und anschließend auf dem MSP430 lauffähig gemacht.

Grundlage für die Umsetzung ist das Pflichtenheft des Reaktionsspiels der Version 0.1. Ansprechpartner und Initiator des Projekts ist Prof. Creutzburg.

## Abgrenzung

Diese Arbeit beschäftigt sich mit Zustandsmaschine des Reaktionsspiels. Es beinhaltet keine Implementierung des kompletten Spiels, stellt aber mit der Zustandsmaschine einen Rahmen für das Spiel bereit und bildet die Grundlage des gesamten Software Designs.

## Pflichtenheft

Aus einem Low Fidelity Erstentwurf, wurde auf Basis des Anforderungshefts ein Pflichtenheft erstellt. Es definiert, wie die Anforderungen realisiert werden sollen und ist in Anhang A: Pflichtenheft zu finden.

# Dokumentation

Mit der Arbeit an dem Projekt sind mehrere Dokumente entstanden, die folgend aufgelistet werden:

* *Projektdokumentation*: Beschreibung der Entwicklungsphasen und des Projektablaufs aus Perspektive des Entwicklers.
* *Pflichtenheft*: Beschreibung der zu entwickelten Lösungen.
* *FSM-Generator*: Enthält das Excelschema, welches die Zustandsmaschine generiert, sowie die zugehörigen drei Quelldateien
* *CSS-Projekt*: lauffähiger Quellcode des Projekts und der Zustandsmaschine

# Entwurf

## Entwicklungsumgebung und Programmiersprache

Mit C als Medium-Level-Language, wird eine maschinennahe und für Mikrocontroller gewöhnliche Sprache genutzt.

Als Entwicklungsumgebung wird Code Composer Studio genutzt, wobei es sich hier um einen Zweig der Entwicklungsumgebung Eclipse handelt, speziell zur Programmierung von Mikrocontrollern von Texas Instruments.

## Der MSP430

Der MSP430 ist ein 16-Bit Controller von Texas Instruments mit sehr geringem Stromverbrauch. Texas Instruments liefert die MSP430 Serie in unterschiedlichen Spezifikationen aus. Dieses Projekt nutzt das MSP-EXP430G2553 LaunchPad.

Erweitert wird der MSP430 durch ein Hardware-Modul, welches aus 2 x (5 Buttons + 5 LED‘s) besteht.

## Zustandsmaschine

State Machines sind ein verbreitetes Konzept bei der Entwicklung von eingebetteten Systemen, womit sich ereignisgesteuerte Systeme modellieren lassen. Mit diesem Konzept lässt sich lesbare und skalierbare Software designen.

Grundsätzlich kann in der Programmiersprache C zur Umsetzung von State Machines die Switch/Case-Anweisung genutzt werden oder aber eine Transition Table. Dieses Projekt nutzt zur Umsetzung letzteres.

Die zu konzipierende Zustandsmaschine stellt die Basis der Softwarearchitektur.

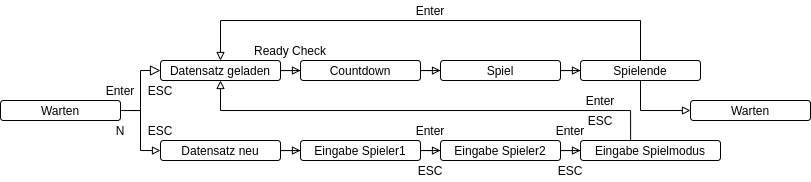
## FSM Generator

Das manuelle Erstellen der Zustandsmaschinen ist fehleranfällig. In diesem Projekt wird ein Code Generation Tool basierend auf Microsoft Excel genutzt. Ist die Zustandsmaschine geplant, so kann hiermit als Folgeschritt, durch Eintragen der Zustände, Events und Zustandsübergangsfunktionen ein modularer C-Programmcode erzeugt werden.

## Benutzerinteraktion

Die Spieler interagieren über eine UART-Schnittstelle des MSP430 auf dialogbasis. Für die Benutzung des Spiels, ist ein lauffähiges Programm “Hyperterm“ auf einem PC notwendig. Des weiteren findet Benutzerinteraktion mit Hilfe des Hardware-Moduls und den darauf implementierten 5 Buttons je Spieler statt.

Die Dialogstruktur kann man sich wie in Abbildung 1 vorstellen.

  
Abbildung 1: Dialogstruktur

## Versionierung

Quellcode und Dokumente werden mit *Git* verwaltet und versioniert. Das Projekt wird auf *Github.com* gehostet und steht öffentlich zur Verfügung.

Das Projekt ist unter folgender URL aufrufbar und klonbar: <https://github.com/JohannesSchumacher/FSM-Reaktionsspiel.git>

## Quelltextdokumentation

Mit doxygen wird die Dokumentation zum Quellcode automatisch generiert. Im CSS-Projektordner ist dafür ein Doxygenkonfigurationsfile namens “*Doxyfile*“ abgelegt, womit eine in HTML generierte Dokumentation erzeugt werden kann, welche ebenfalls im CSS-Projektordner namens “html“ zu finden ist.

## Diagramme

Erstellte Diagramme, wie die Dialogstruktur in Abbildung 1: Dialogstruktur oder das Zustandsdiagramm, sind im Doku-Verzeichnis des Projekts im .drawio-Format hinterlegt. Sie können neben der Dokumentation auch Online, unter der URL [www.draw.io](http://www.draw.io/), geöffnet und bearbeitet werden.

## Testfälle

Zu testen ist ob, die Zustandsabfolge korrekt ist und bestimmte Zustände korrekt erreicht werden. Außerdem wird getestet, ob die Zeiger auf die Zustandsübergangsfunktionen korrekt sind und die Funktionen aufgerufen werden.

Um die Testfälle zu prüfen, wird das CSS-Projekt genutzt und Programmausdrücke darin verfolgt und debuggt.

# Durchführung

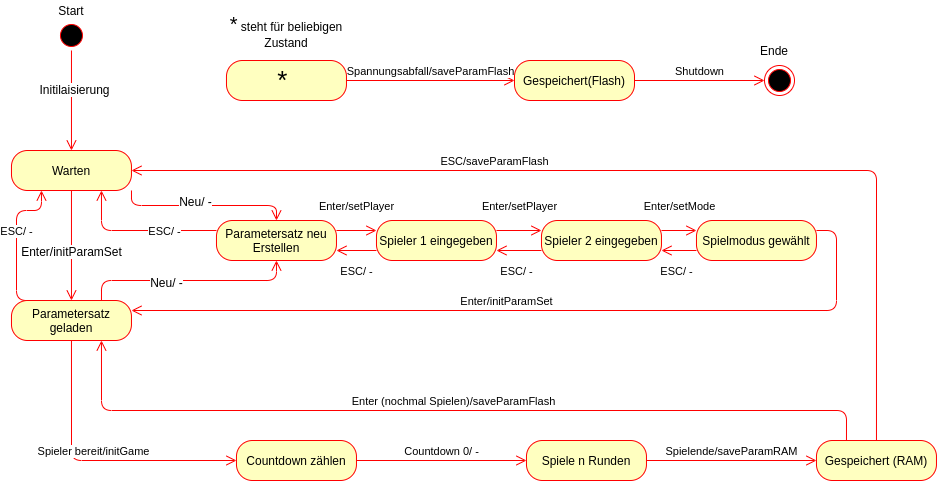
## Ausgangslage

Die Zustandsmaschinen, wie sie im Pflichtenheft bereits veröffentlicht wurden, sollen als erster Ausgangsentwurf dienen. Durch einen iterativen Entwicklungsprozess, folgt eine Detaillierung.

## Die Zustandsmaschine im Überblick

### **Erstellung**

Die Erwägungen des Pflichtenhefts bezüglich der Zustandsmaschinen wurden, auch nach Rücksprache mit Prof. Creutzburg, voll einbezogen. Es wurde versucht einen Kompromiss aus Detaillierungsgrad der Zustände des Spiels und geringer Komplexität/Übersichtlichkeit zu finden. Die Bezeichner der Zustände und Events wurden eindeutiger und zusätzlich um Funktionsbezeichner in der Form *Event/C-Funktion* ergänzt.

  
Abbildung 2: Zustandsdiagramm

Die Zustandsmaschine in Abbildung 2 behandelt Funktionalitäten, die in 3 Teilbereiche gegliedert werden können:

1. **Parametersatzerstellung und Verwaltung**: Das Programm arbeitet zur Laufzeit mit einem Parametersatz, der in C mittels *typedef* als Struktur deklariert ist. Die Zustandsmaschine berücksichtigt das Laden eines bereits vorhanden Parametersatzes aus dem Flash, als auch das Neuerstellen eines Parametersatzes. Ist kein Parametersatz im Flash hinterlegt, kann dies programmatisch geprüft werden und als Folgerung ein default Parametersatz initialisiert werden. Wird ein Parametersatz neu erstellt, so werden 2 Spieler und ein Spielmodus erfragt. Sind die Benutzereingaben vollendet, wird ein neuer Parametersatz initialisiert.
2. **Spielhinführung und Ablauf:** Ist der Parametersatz geladen, kommt es zum Reaktionsduell. Ein Event, indem sich beide Spieler (durch beispielsweise eines Buttondrucks) auf ihrem Hardware-Modul bereit zeigen, initiiert einen Countdown und anschließend das Spiel
3. **Speicherung sowohl in RAM als auch Flash:** Ist das Spiel abgeschlossen, werden Informationen wie etwa Gewinner, Spielstand und Reaktionsstatistiken im Parametersatz gespeichert. Hierbei ist zu unterscheiden. Das Spiel arbeitet zur Laufzeit mit dem Parametersatz auf dem flüchtigen RAM, welcher zur Laufzeit schnellen Speicherzugriff erlaubt. Nur zu dezidierten Zeitpunkten speichert das Spiel den Parametersatz persistent im Flash-Speicher. Dies Passiert dann, wenn ein Spannungsabfall erkannt wird oder nach regulärem Spielende.

### Die Zustände

*Tabelle 1* beschreibt alle Zustände detailliert. Quellcodebezeichner sind die für den Quellcode gedachten englischen Kurzbegriff.

Tabelle 1: Zustände

|  |  |  |
| --- | --- | --- |
| **Zustand** | **Quellcodebezeichner** | **Beschreibung** |
| Warten | Wait | * Erster Zustand nach Initialisierung des Controllers |
| Parametersatz neu Erstellen | Build\_ParamSet | * Neuerstellung eines Parametersatzes |
| Spieler 1 eingegeben | Set\_Player1 | * Zustand nach der Eingabe des ersten Spielernamen |
| Spieler 2 eingegeben | Ser\_Player2 | * Zustand nach der Eingabe des zweiten Spielernamen |
| Spielmodus gewählt | Set\_Gamemode | * Zustand nach Auswahl des Spielmodi |
| Parametersatz geladen | Load\_ParamSet | * Erreichter Zustand sobald ein neuer oder bereits vorhandener Zustand geladen ist |
| Countdown zählen | Countdown | * Countdown für den Spielstart |
| Spielen n Runden | Play | * Beide Spieler spielen das Spiel |
| Gespeichert (RAM) | Save\_RAM | * Spiel und Statistik wurde im ParamSet auf RAM gespeichert |
| Gespeichert (Flash) | Save\_Flash | * ParamSet wurde auf dem Flashspeicher hinterlegt |

### Die Events

Zustandstransitionen durch Events werden in *Tabelle 2* detailliert beschrieben.

Tabelle 2: Events

|  |  |  |
| --- | --- | --- |
| **Event** | **Quellcodebezeichner** | **Beschreibung** |
| Enter | CR | * Event zum Bestätigen von Eingaben und Einstellungen |
| ESC | ESC | * Event zur Rückgängigmachung von Eingaben und Einstellungen |
| Neu | N | * Leitet die Erstellung eines neuen Parametersatzes ein |
| Spieler bereit | Ready | * Beide Spieler bestätigen, dass sie bereit für das Spiel sind |
| Coundown 0 | Countdown0s | * Das Spiel zählt eine Startampel an |
| Spielende | Win | * Ein Spieler hat das Spiel gewonnen oder das Spielziel wurde erreicht |
| Spannungsabfall | Voltagedrop | * Es wird einen Spannungsabfall registriert |

### Die Zustandsübergangsfunktionen

Die Events sind vereinzelt mit Funktionen verknüpft, die beim eintreten des jeweiligen Events aufgerufen werden. Die Eingangsfunktion *Initialisierung, zu sehen* am Start des Zustandsiagramms in Abbildung 2: Zustandsdiagramm, sowie die Beendenfunktion *Shutdown* am Endpunkt des Diagramms in Abbildung 2, werden nicht aufgeführt und sollten extra implementiert werden. *Tabelle 3* beschreibt die Funktionsübergangsaufrufe.

Tabelle 3: Transitionsaufrufe

|  |  |  |
| --- | --- | --- |
| **Transitionsfunktion** | **Quellcodebezeichner** | **Beschreibung** |
| initParamSet | initParamSet | * Funktion zum laden des Parametersatzes in RAM |
| setPlayer | setPlayer | * Setzt einen Spielernamen |
| setMode | setMode | * Setzt den Spielmodus |
| initGame | initGame | * Initiiert das Zählen des Countdowns und initialisiert das gewählte Spiel |
| saveParamRAM | saveParamRAM | * Speichert/Aktualisiert den Parametersatz im RAM |
| saveParamFlash | saveParamFlash | * Überträgt den Parametersatz vom RAM auf den Flashspeicher |

## FSM Quellcodegenerator

Die Zustandsmaschine Abbildung 2: Zustandsdiagramm wurde in das FSM-Excel-Schema von Texas Instruments übertragen. Zur Generierung des Quellcodes wurden lediglich, statt den deutschen Bezeichnern, englische Bezeichner für Zustände und Events gewählt. In Anhang B: FSM-Excel-Schema ist die Exceltabelle zu finden. Der generierte Quellcode befindet sich in Anhang C: Generierter Quellcode der FSM.

Es wurden 3 Dateien generiert:

* *fsm.h*: Das sogenannte Header File enthält alle Bekanntmachungen der Zustandsmaschine von Transition Table bis hin zu den Aliasen von Zuständen und Events, sowie den Funnktionen der Eventaufrufe und Transitionen. Sie muss in jeder Quelldatei eingebunden werden, die mit der Zustandsmaschine arbeitet.
* *fsm.c*: Diese Datei beinhaltet die Implementierung der Zustandstabelle und Eventfunktionen. Bei Auftreten des Events müssen die korrespondierenden Eventfunktionen aufgerufen werden.
* *fsm\_transition.c*: Hier sind die Zustandsübergangsfunktionen generiert und implementiert, welche über einen Zeiger in der Eventsfunktion aufgerufen werden.

## Implementierung mit Code Composer Studio

### Projekteinrichtung

Zum Entwickeln und Testen wurde eine virtuelle Maschine mit Oracle Virtualbox eingerichtet. Es wurde außerdem eine Target, der MSP430G2553, als Zielplattform eingerichtet. Das CCS-Projekt wurde erstellt und die generierten C- und Header-Quelldateien der FSM in das Projekt eingebunden, sowie ein main.c-File erstellt, wo die Zustandsmaschine eingebunden wurde. Um Komplikationen beim Kompilieren zu vermeiden, mussten die Präprozessor-defines des Zustands “*Wait*“ und des Events „“*N*“ umbenannt werden. Wait wurde in *Init* umbenannt, bzw. das Event N in *Neu.* Dies geschah respektive in den Event-Funktionen und der Transitionstabelle, wo mit den Zuständen und Events gearbeitet wird.

### Event-Aufrufe in der main.c

Nach Einrichtung und Anpassung des CCS-Projekts, wurden alle Event-Aufrufe, die zu einem Zustandsübergang führen, getestet. Die Funktionen für die Event-Aufrufe befinden sie in der fsm.c und wurden in der main.c aufgerufen.

Die extern (in fsm.h) deklarierte Variable *ActState, die den aktuellen Zustand darstellt,* wurde beobachtet und als Expression im Debugger hinterlegt. Die Event-Aufrufe wurden mit Kommentaren versehen, die Beschreiben was im Spielablauf oder Programm nun abstrakt passiert. (siehe dazu D.1 ActState)

### Nachvollziehbarkeit der Events

Zum Testen und Nachvollziehen der Zustandsübergange und Zustandsübergangsfunktionen, wurde im Debug-Modus geprüft ob alle Programmpunkte erreicht werden. Hierbei konnte der *ActState* alle Zustände korrekt annehmen, nachdem ein Event-Aufruf getätigt wurde. Außerdem wurden die Zustandsübergangsfunktionen in fsm\_transition.c aufgerufen. D.h. also, dass Zeiger auf diese Funktionen korrekt sind. Zu sehen ist das auch am Sprung in den Assemblercode, wo mit der Adresse von Funktionen im Speicher gearbeitet wird. (siehe D.2 Zeiger auf Funktionsaufruf)

### Generierung der Quellcodedokumentation

Abschließend wurde eine Quellcodedokumentation mit Doxygen generiert, die die Datenstrukturen der Statetable und Funktionsaufrufe über das HTML-Format einsehbar machen.

# Funktionales Testen

Im wesentlichem gingen Tests unmittelbar mit der Implementierung einher. Nach wenigen Entwicklungsschritten wurde das Programm auf Funktionalität getestet und ein kompilierungsfähiger Stand geschaffen, der anschließend debuggt wurde. Hierbei wurde speziell die externe Variable *ActState* als Ausdruck verfolgt, aber auch das korrekte Zeigen auf die Transitionsfunktionen getestet, wie in D.1 ActState und D.2 Zeiger auf Funktionsaufruf auch zu sehen ist.

# Auswertung

## Erreichen des Projektziels

Das Projektziel konnte, nach den Eingangs in diesem Dokument erklärten Zielformulierungen, sowie auch die Zielformulierungen im Pflichtenheft, erreicht werden. Aufgrund der selbstständigen Arbeit an dem Gruppenprojekt, wurden Kompromisse eingegangen. Es wurde sich vor allem, auf die Zustandsmaschine beschränkt. Mit den Gedanken zur Entwicklung einher, ging eine konzeptionelle Ausrichtung und Planung des Gesamtprojekts, wobei die Zustandsmaschine diese Gedanken abbildet und als Fundament der Softwarearchitektur des Projekts aufgefasst werden kann. Des weiteren flossen Gedanken zur Benutzerinteraktion und einer kohärenten Dialogstruktur ein, um ein (soweit möglich) intuitives Benutzererlebnis zu erlauben. Auf Basis der Zustandsmaschine sind nun einige Folgeimplementierungen möglich. Es wäre nun spannend, zu sehen, wie robust die FSM in einem komplett implementierten Spiel nach Anforderungskatalog funktioniert und ob eventuelle weitere Berücksichtigungen in die Konzeption der FSM mit einbezogen werden hätten können.

## Folgeentwicklungen

### Entwicklung der Dialogstruktur mit UART

Mit UART können große Teile der Benutzerinteraktion realisiert werden. Hierzu gehören Benutzereingaben, die in der Zustandsmaschine beschrieben sind, als auch Infoausgaben und Programmrückmeldungen. Die Eingaben sollten mit den Transitionsfunktionen *setName* und *setMode* verknüpft werden.

### Parametersatz

Die Datenstruktur für den Parametersatz *ParamSet,* mit all seinen enthaltenen Informationen zum Spiel, Spielstatistik und Programmstatistik ist fundamental. Hierzu gehört das Speichern und Laden des Parametersatzes aus dem Flash, sowie die CRC Prüfimplementierung. Das Laden und Erstellen des Parametersatzes ist in der Zustandsmaschine berücksichtigt und sollte damit verknüpft werden.

### Implementierung des Spiels

Die eigentliche Implementierung des Reaktionsspiels ist notwendig. Die Zustandsmaschine stellt Funktionen zur Initialisierung des Spiels, einem Bereitschaftscheck und einen Countdown. Spielergebnisse werden anschließend im Parametersatz gespeichert, was auch in der FSM berücksichtigt ist.

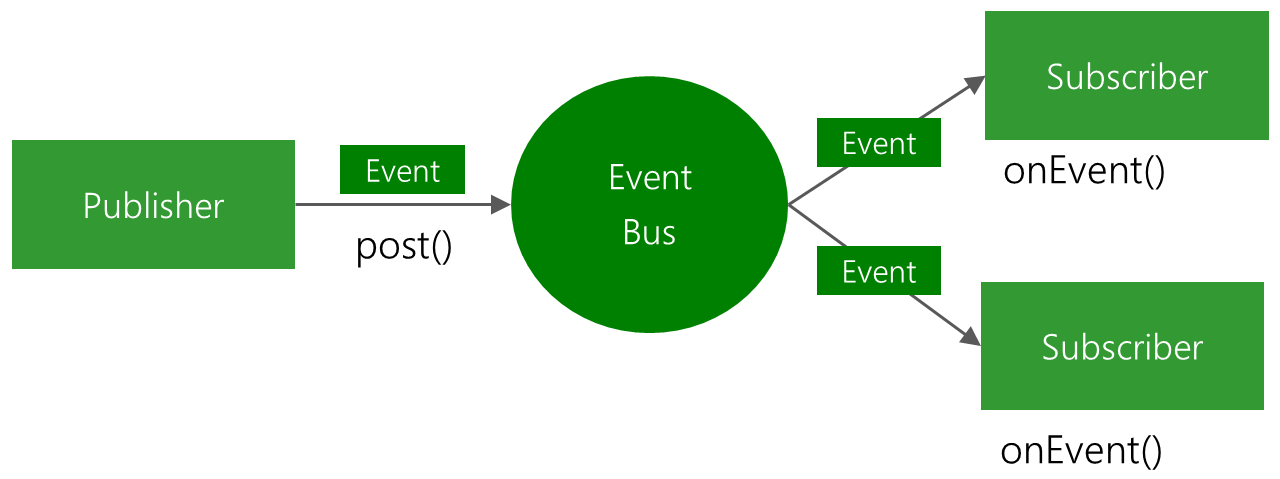
### Ausnahmebehandlungen

Was passiert wenn die Spannungsversorgung einbricht? Wie lange hält die Stromversorgung danach noch an? Diese Extremfälle müssen berücksichtigt werden und Behandlungen implementiert werden.

## Erkenntnisse

Das Beschäftigen mit der methodische Entwickeln einer FSM und den Implementierungsmöglichkeiten in C ging ein neuer Erfahrungsgewinn einher. Das Wissen zum Entwickeln von eventgesteuerter Software ist vielseitig einsetzbar. Neben der großen Relevanz im Bereich Embedded Systems, ist ein Event-Konzept z.B auch im Bereich der Android-Entwicklung zu finden ( Abbildung 3: Eventbus für Android).

Ebenso ist im Bereich des ROS ein ausgereiftes Publish und Subscribe-System für verteilte Systeme zu finden, indem Kommunikation mit einem Eventhandlingsystem stattfindet.

  
Abbildung 3: Eventbus für Android

Die Idee der Events und des Eventhandlings ist also nahezu überall als Konzept in der Softwareentwicklung vorhanden und speziell in diesem Projekt war das Umsetzen der FSM eine grundlegend praktische und prägende Erfahrung.

# Anhang A: Pflichtenheft

Anfang Pflichtenheft

**Allgemeines**

**Dokument**

Dieses Pflichtenheft beschreibt die Umsetzung des Reaktionsspiels des Moduls Embedded Systems für den Studiengang SMSB nach dem Dokument „Anforderungen Reaktionsspiel“ von Prof. Creutzburg.

**Projektbezug**

In Bezug auf das Projekt und den Anforderungskatalog, wird hier speziell die Konzeption, Generierung und Umsetzung möglicher Zustandsmaschinen beschrieben, die zur strukturierten Umsetzung des Reaktionsspiels betrachtet werden können. Es ist als Teilprojekt des Reaktionsspiels zu sehen, wessen Grundlage die Liste der zu implementierenden Features aus dem Anforderungsdokument ist.

Im folgenden Verlauf werden die wesentlichen Kriterien des Anforderungsdokuments zur Umsetzung der Zustandsmaschinen herausgearbeitet.

**Ziel und Idee**

Dieses Dokument beschreibt die Konzeption, Generierung und Umsetzung von Zustandsmaschinen auf dem TI MSP430 Launch Pad. Es beschreibt eine mögliche Umsetzung des Reaktionsspiels im Rahmen des Moduls Embedded Systems und dient als Lernerfahrung des Studenten im praktischen Umsetzen von Zustandsmaschinen nach einem strukturierten Entwurfsmuster. Hierzu gehört die Planung und Konzeption mit anschließender Generierung des C-Programmcodes mit dem FSM Generator von Texas Instruments. Zuletzt folgt eine Testweise Implementierung einer Zustandsmaschine auf dem MSP430, wobei die Funktionsfähigkeit durch das Verfolgen der Zustände nachvollzogen werden kann.

**Features**

**Musskriterien**

Zum Spielablauf:

* Die Spieler geben ihre Namen ein
* Die Spieler haben verschiedene Spielmodi zur Auswahl
* Die Spieler können ein ausgewähltes Reaktionsspiel spielen

Zum Programm:

* Das Programm arbeitet mit einem Parametersatz
* Der Parametersatz enthält verschiedene Daten wie Spielernamen, gewählter Spielmodi, statistische Aussagen, Betriebsspannungen
* Der Parametersatz wird nach Beenden des Spiels(Spannungsabfall) persistent auf dem Flash abgelegt
* Zur Laufzeit wird mit einer Kopie des Parametersatzes im RAM gearbeitet

Verschiedenes:

* Einhaltung von Programmierstandards ([SEI CERT C Coding Standard](https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard), MISRA-C Regeln)
* Quellcodedoku (Doxygen)
* Modularisierung (Header/C-Dateien)
* Verwendung des Excel-Code-Generators für State Machines
* Versionsverwaltung (Git)
* Git-Remote-Host mit Gitlab oder Github
* Abhängige Dateien wie lokale Buildabhängigkeiten werden nicht versioniert (gitignore)

**Wunschkriterien**

Zum Spielablauf:

* Die Spieler können sich wiederholt duellieren ohne auf redundante Eingabeaufforderung zu stoßen
* Es wird im wesentlichen auf eine intuitive Benutzbarkeit wert gelegt

**Abgrenzungskriterien**

* Spieleranzahl: 2
* Die aufgelisteten Anforderungen sind eine Abstraktion um relevante Kriterien für die Entwicklung der State Machines (als Teilprojekt) hervorzuheben
* Fokus ist das Entwickeln der State Machines
* Es findet keine Implementierung der vollständigen Spiellogik statt

**E****insatz**

**Anwendungsbereiche**

Zwei spiellustige Menschen können sich jederzeit in dem Reaktionsspiel duellieren, wobei es um schnelles Erkennen und Reagieren geht.

**Zielgruppe**

Jeder kann das Spiel spielen.

**Betriebsbedingungen**

Zum Spielen benötigt wird ein MSP430 mit erweiterter Hardware-Plattform (Taster & LEDs). Des weiteren wird zur Kommunikation mit den Spielern ein Host-Rechner benötigt sowie ein Terminal-Programm.

**Projektumgebung**

Die Lösung ist auf dem MSP-EXP430G2 LanuchPad lauffähig. Das Programm ist über eine serielle Schnittstelle mit einem Host-Rechner installierbar und auf dem MSP430 ausführbar.

**Spielfunktionen**

**/F10/ Setzen der Spieler und Wahl des Spielmodus**

* /F11/ Eingabe Spieler 1: Benutzereingabe über UART-Schnittstelle und Speicherung im Parametersatz (RAM)
* /F12/ Eingabe Spieler 2: Benutzereingabe über UART-Schnittstelle und Speicherung im Parametersatz (RAM)
* /F13/ Spielmodus wählen über UART-Schnittstelle und Speicherung im Parametersatz (RAM)

**/F20/ Spielablauf**

* /F21/ Bereitschaftscheck: Beide Spieler bestätigen durch einen Ready-Check
* /F22/ Countdown: Das Spiel zählt einen Countdown herunter (Darstellung mit LEDs)
* /F23/ Das Spiel: Die Spieler spielen eine gegebene Rundenzahl
* /F24/ Spielende: Speichere Statistiken in Parametersatz im RAM

**/F30/ Beenden**

Das Programm speichert den Parametersatz im Flash und erreicht einen Anfangszustand

**Nicht-funktionale Anforderungen**

**Erweiterbarkeit**

Die Zustandsmaschinen können zur vollständigen Implementierung des Reaktionsspiels herangezogen werden und spezialisiert werden. Neu entwickelte Spielmodi können hinzugefügt werden.

**Programmiersprache**

Es wird mit der Programmiersprache C und den MSP430 Libraries unter Einhaltung der Programmierstandards gearbeitet.

**Programmierumgebung**

Es wird mit Code Composer Studio als Entwicklungsumgebung programmiert.

**Modifizierung**

Das Projekt wird zur Modifizieren und Anpassung zur Verfügung gestellt und remotehostet.

**Spield****aten**

**/D10/ Benutzerdaten**

Gemäß /F11/ und /F12/ geben die Benutzer ihre Spielnamen ein:

* /D11/ Spielname 1
* /D12/ Spielname 2

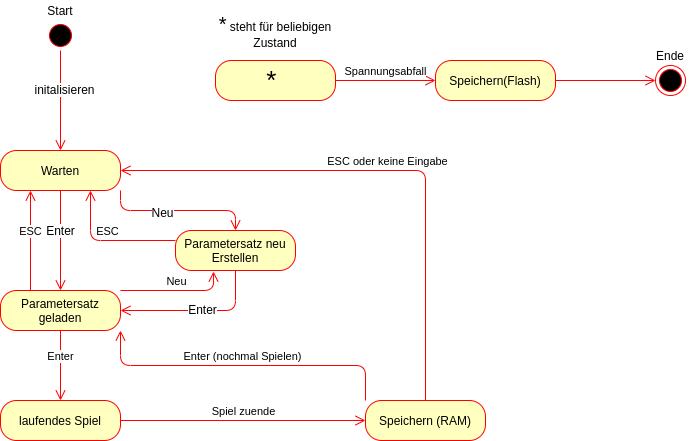
**/D20/ Statistiken**

Mit dem Spielende (Funktion /F24/) werden Statistiken gespeichert wie Spielstand, durchschnittliche Reaktionszeit und Anzahl Gewinner links/rechts.

**Zustandsmaschinen**

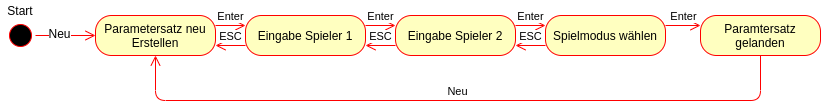
**Controller-****Hauptzustandsmaschine**

Es folgt erste Konzeption der Hauptzustandsmaschine, die im Laufe der Entwicklung vervollständigt und angepasst wird bzw. die Events mit Funktionsaufrufen verknüpft werden.



**Datensatzauswahl und Spielereingaben**

Ausgehend vom Event *Neu* und dem Zustand *Parametersatz neu Erstellen* (siehe Hauptzustandsmaschine) kann das Neuerstellen des Parametersatzes zustandsmaschinenbasiert passieren.



**Spielablauf**

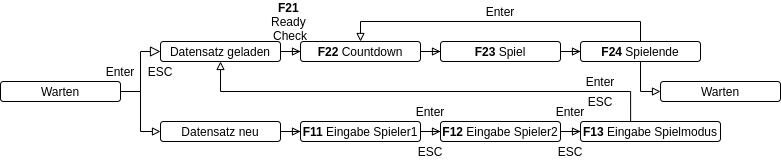
Auch der Spielfluss kann zustandsbasiert konkretisiert werden. Ausgangspunkt ist der Zustand *Parametersatz geladen* (siehe Hauptzustandsmaschine).



**Benutzer****interaktion**

**Dialogstruktur**

Im Folgenden wird die Dialogstruktur einer fehlerfreien bzw. konfliktfreien Benutzung des Spiels gezeigt. Fehlereingaben haben im Zweifel einen Rücksprung auf den Anfangszustand zur Folge. Mit **F??** wird auf die Spielfunktionen (aus Kapitel 5) referenziert.



N

**Benutzerinput**

Die Benutzer navigieren in der Regel mit **Enter** (Vor )und **ESC** (Zurück) durch die Dialogstruktur. Ausnahme ist das setzen eines neues Datensatzes, das mit **N** für Neu/New passiert.

**Qualitätsanforderungen**

Auf folgende Qualitätsstandards wird besonders Wert gelegt:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *sehr wichtig* | *wichtig* | *weniger wichtig* | *unwichtig* |
| *Zuverlässigkeit* | × |  |  |  |
| *Geringe Komplexität* | × |  |  |  |
| *Ressourceneffizienz* |  | × |  |  |
| *Benutzerfreundlich* |  |  | × |  |

**Testszenarien**

Grundsätzlich folgt auf jede funktionale Anforderung **/F??/** ein konkreter Testfall **/T??/** . Es folgt ein konkretes Testszenario des fehlerfreien Spielablaufs:

* /T11/ Eingabe Spieler 1: Der Testspieler 1 wählt mit N wie neu einen neuen Datensatz aus. Er gibt seinen Namen ein und bestätigt mit Enter.
* /T12/ Eingabe Spieler 2: Der Testspieler 2 gibt ebenso seinen Spielnamen an und bestätigt mit Enter.
* /T13/ Eingabe Spielmodi: Ein Testspieler wählt mit einer Zahl einen Spielmodus aus einer gegebenen Spielliste mit der korrespondierenden Listenzahl als Spiels. Die Eingabe wird geprüft.
* /T21/ Bereitschaftscheck: Die Testspieler bestätigen ihre Bereitschaft mit einem Knopfdruck auf jeweils einen ihrer Taster.
* /T22/ Countdown: Das Spiel zählt von selbst einen Countdown herunter, in Form von ausgehenden LEDs.
* /T23/ Das Spiel: Testspieler 1&2 spielen das Reaktionsspiel n Runden.
* /T24/ Spielende: Durch keine Eingabe oder ESC geht das Spiel in den Anfangszustand.
* /T30/ Beenden: Ein Testbenutzer löst einen Spannungsabbruch aus.

Ende Pflichtenheft

# Anhang B: FSM-Excel-Schema



# Anhang C: Generierter Quellcode der FSM

## C.1 fsm.h

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// MSP430 state machine

// fsm.h

//

// Describtion:

// A simple state machine for the MSP430

//

// Generated with Excel Table

// Date: 02/11/2020 Time: 21:56:52

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

#ifndef FSM\_H

#define FSM\_H

#include <stdint.h>

#include <stdio.h>

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// State table typedef

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

typedef struct

{

void (\*ptrFunct) (void);

uint8\_t NextState;

} FSM\_STATE\_TABLE;

extern uint8\_t ActState;

#define NR\_EVENTS 7

#define CR 0

#define ESC 1

#define N 2

#define READY 3

#define COUNTDOWN0S 4

#define WIN 5

#define VOLTAGEDROP 6

#define NR\_STATES 10

#define WAIT 0

#define BUILD\_PARAMSET 1

#define SET\_PLAYER1 2

#define SET\_PLAYER2 3

#define SET\_GAMEMODE 4

#define LOAD\_PARAMSET 5

#define COUNTDOWN 6

#define PLAY 7

#define SAVE\_RAM 8

#define SAVE\_FLASH 9

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Function prototypes

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Initialize state machine

void FSM\_Init (void);

// Event function "CR"

void FSM\_CR (void);

// Event function "ESC"

void FSM\_ESC (void);

// Event function "N"

void FSM\_N (void);

// Event function "Ready"

void FSM\_Ready (void);

// Event function "Countdown0s"

void FSM\_Countdown0s (void);

// Event function "Win"

void FSM\_Win (void);

// Event function "Voltagedrop"

void FSM\_Voltagedrop (void);

// Transition function "initParamSet"

void FSM\_initParamSet (void);

// Transition function "saveParamFlash"

void FSM\_saveParamFlash (void);

// Transition function "setPlayer"

void FSM\_setPlayer (void);

// Transition function "setMode"

void FSM\_setMode (void);

// Transition function "initGame"

void FSM\_initGame (void);

// Transition function "saveParamRAM"

void FSM\_saveParamRAM (void);

// Transition function "init"

void FSM\_init (void);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

#endif /\* FSM\_H \*/

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

## C.**2** fsm.**c**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// MSP430 state machine

// fsm.c

//

// Describtion:

// A simple state machine for the MSP430

// Do not change code in here!!!

//

// Generated with Excel Table

// Date: 02/11/2020 Time: 21:59:27

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

#include <stdint.h>

#include <stdio.h>

#include "fsm.h"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Global variables

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

uint8\_t ActState;

const FSM\_STATE\_TABLE StateTable [NR\_STATES][NR\_EVENTS] =

{

FSM\_initParamSet, LOAD\_PARAMSET, NULL, WAIT, NULL, BUILD\_PARAMSET, NULL, WAIT, NULL, WAIT, NULL, WAIT, FSM\_saveParamFlash, SAVE\_FLASH,

FSM\_setPlayer, SET\_PLAYER1, NULL, WAIT, NULL, BUILD\_PARAMSET, NULL, BUILD\_PARAMSET, NULL, BUILD\_PARAMSET, NULL, BUILD\_PARAMSET, FSM\_saveParamFlash, SAVE\_FLASH,

FSM\_setPlayer, SET\_PLAYER2, NULL, BUILD\_PARAMSET, NULL, SET\_PLAYER1, NULL, SET\_PLAYER1, NULL, SET\_PLAYER1, NULL, SET\_PLAYER1, FSM\_saveParamFlash, SAVE\_FLASH,

FSM\_setMode, SET\_GAMEMODE, FSM\_setPlayer, SET\_PLAYER1, NULL, SET\_PLAYER2, NULL, SET\_PLAYER2, NULL, SET\_PLAYER2, NULL, SET\_PLAYER2, FSM\_saveParamFlash, SAVE\_FLASH,

FSM\_initParamSet, LOAD\_PARAMSET, FSM\_setPlayer, SET\_PLAYER2, NULL, SET\_GAMEMODE, NULL, SET\_GAMEMODE, NULL, SET\_GAMEMODE, NULL, SET\_GAMEMODE, FSM\_saveParamFlash, SAVE\_FLASH,

NULL, LOAD\_PARAMSET, NULL, LOAD\_PARAMSET, NULL, LOAD\_PARAMSET, FSM\_initGame, COUNTDOWN, NULL, LOAD\_PARAMSET, NULL, LOAD\_PARAMSET, FSM\_saveParamFlash, SAVE\_FLASH,

NULL, COUNTDOWN, NULL, COUNTDOWN, NULL, COUNTDOWN, NULL, COUNTDOWN, NULL, PLAY, NULL, COUNTDOWN, FSM\_saveParamFlash, SAVE\_FLASH,

NULL, PLAY, NULL, PLAY, NULL, PLAY, NULL, PLAY, NULL, PLAY, FSM\_saveParamRAM, SAVE\_RAM, FSM\_saveParamFlash, SAVE\_FLASH,

FSM\_saveParamFlash, LOAD\_PARAMSET, FSM\_saveParamFlash, WAIT, NULL, SAVE\_RAM, NULL, SAVE\_RAM, NULL, SAVE\_RAM, NULL, SAVE\_RAM, FSM\_saveParamFlash, SAVE\_FLASH,

NULL, SAVE\_FLASH, NULL, SAVE\_FLASH, NULL, SAVE\_FLASH, NULL, SAVE\_FLASH, NULL, SAVE\_FLASH, NULL, SAVE\_FLASH, FSM\_init, WAIT

};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Initialize state machine

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_Init (void)

{

ActState = WAIT;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Event function "CR"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_CR (void)

{

if (StateTable[ActState][CR].ptrFunct != NULL)

StateTable[ActState][CR].ptrFunct();

ActState = StateTable[ActState][CR].NextState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Event function "ESC"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_ESC (void)

{

if (StateTable[ActState][ESC].ptrFunct != NULL)

StateTable[ActState][ESC].ptrFunct();

ActState = StateTable[ActState][ESC].NextState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Event function "N"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_N (void)

{

if (StateTable[ActState][N].ptrFunct != NULL)

StateTable[ActState][N].ptrFunct();

ActState = StateTable[ActState][N].NextState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Event function "Ready"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_Ready (void)

{

if (StateTable[ActState][READY].ptrFunct != NULL)

StateTable[ActState][READY].ptrFunct();

ActState = StateTable[ActState][READY].NextState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Event function "Countdown0s"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_Countdown0s (void)

{

if (StateTable[ActState][COUNTDOWN0S].ptrFunct != NULL)

StateTable[ActState][COUNTDOWN0S].ptrFunct();

ActState = StateTable[ActState][COUNTDOWN0S].NextState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Event function "Win"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_Win (void)

{

if (StateTable[ActState][WIN].ptrFunct != NULL)

StateTable[ActState][WIN].ptrFunct();

ActState = StateTable[ActState][WIN].NextState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Event function "Voltagedrop"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_Voltagedrop (void)

{

if (StateTable[ActState][VOLTAGEDROP].ptrFunct != NULL)

StateTable[ActState][VOLTAGEDROP].ptrFunct();

ActState = StateTable[ActState][VOLTAGEDROP].NextState;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

## C.**3** fsm\_transition.c

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// MSP430 state machine

// fsm\_transition.c

//

// Describtion:

// A simple state machine for the MSP430

// You can add your own code in here!!!

//

// Generated with Excel Table

// Date: 02/11/2020 Time: 21:54:49

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

#include "fsm.h"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Transition function "initParamSet"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_initParamSet (void)

{

// You can add your code here

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Transition function "saveParamFlash"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_saveParamFlash (void)

{

// You can add your code here

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Transition function "setPlayer"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_setPlayer (void)

{

// You can add your code here

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Transition function "setMode"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_setMode (void)

{

// You can add your code here

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Transition function "initGame"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_initGame (void)

{

// You can add your code here

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Transition function "saveParamRAM"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_saveParamRAM (void)

{

// You can add your code here

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

// Transition function "init"

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

void FSM\_init (void)

{

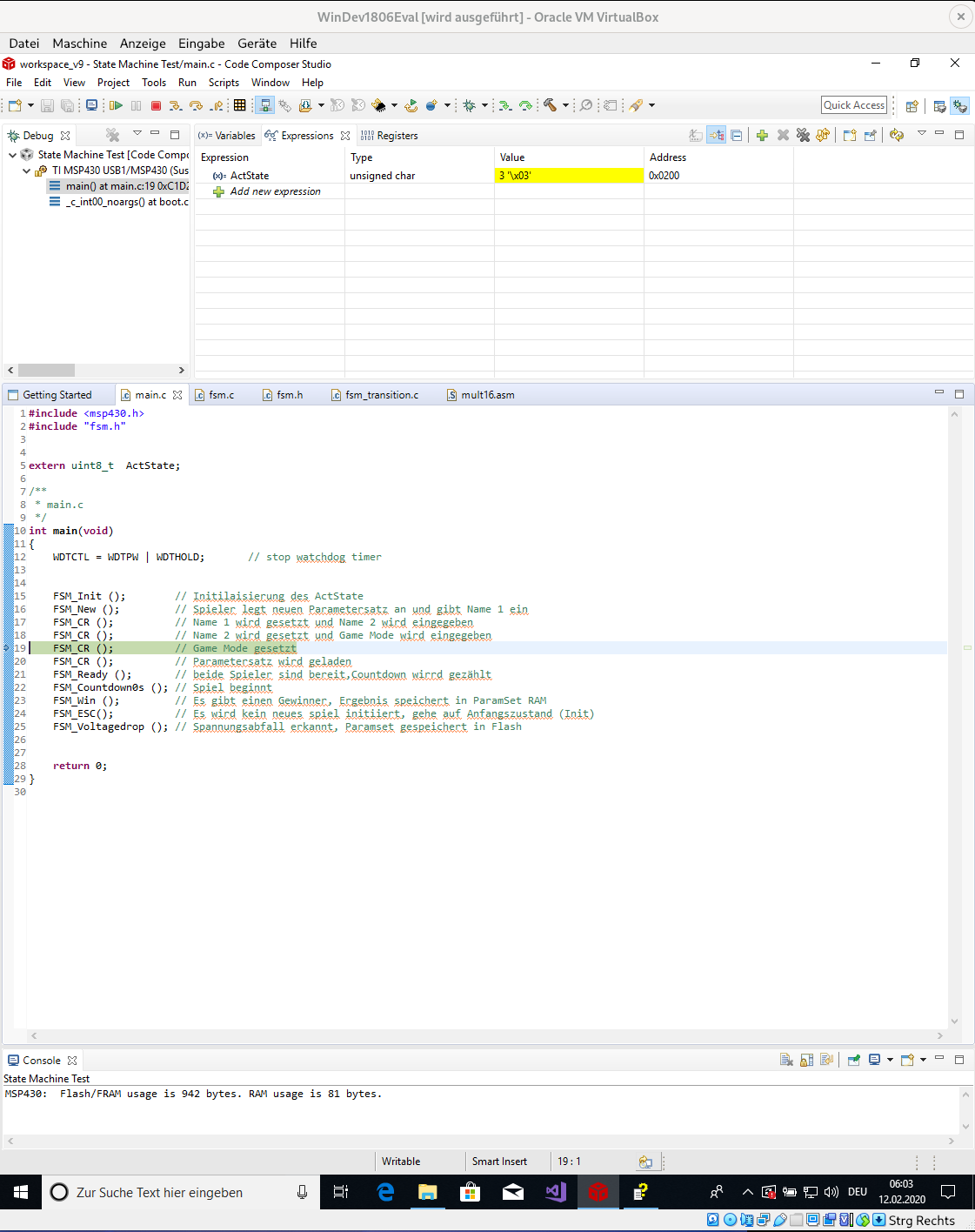
// You can add your code here

}

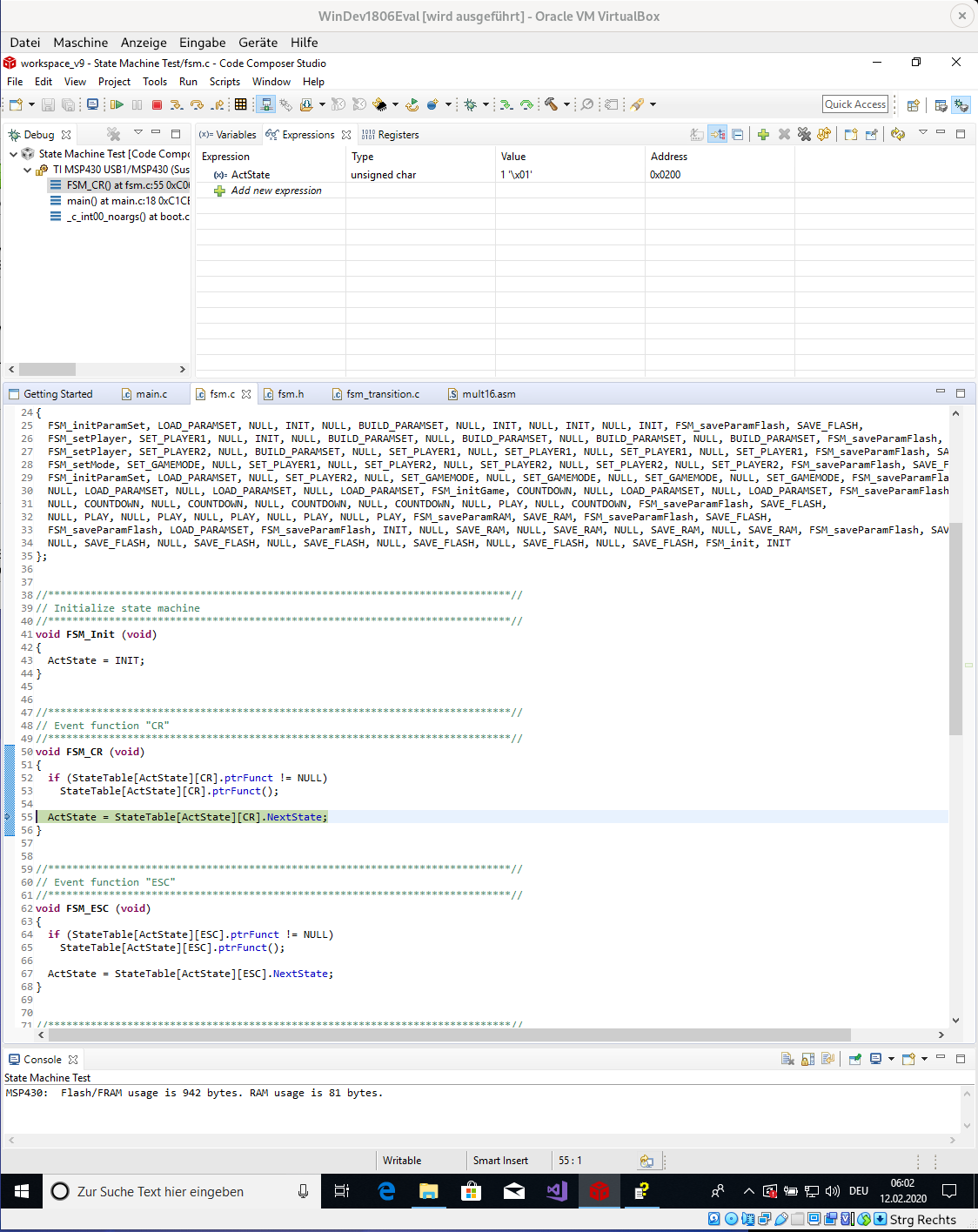
//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

# Anhang D: Testen des Zustands und der Übergänge

## D.1 ActState



## D.2 Zeiger auf Funktionsaufruf



# Quellenverzeichnis

**RobinH107**: Finite State Machine on a MSP430. <https://www.instructables.com/id/Finite-State-Machine-on-a-MSP430/> (Datum des Zugriffs: 12. Februar 2020).

**Texas Instruments** (2009): Finite State Machines for MSP430. <http://www.ti.com/lit/an/slaa402a/slaa402a.pdf> (Datum des Zugriffs: 12. Februar 2020).