

# Implementierung einer interruptgesteuerten Benutzerschnittstelle auf einem Low-Power-Mikrocontroller

## Bachelor-Thesis

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.) im Studienfach Angewandte Informatik



**Hochschule Konstanz**  
Fakultät Informatik

vorgelegt von: Julian Rapp

Matrikelnummer: 304875

Erstgutachter: Prof. Dr. Irenäus Schoppa

Zweitgutachter: Prof. Dr. Heiko von Drachenfels

eingereicht in: Konstanz, am 30. Juni 2025

# Eidesstattliche Erklärung

Hiermit erkläre ich, Julian Rapp, geboren am 07.03.2001 in Freudenstadt,

1. dass ich meine Bachelor-Thesis mit dem Titel:

**„Implementierung einer interruptgesteuerten Benutzerschnittstelle  
auf einem Low-Power-Mikrocontroller“**

in der Fakultät Informatik unter Anleitung von Herrn Prof. Dr.

Irenäus Schoppa selbständig und ohne fremde Hilfe angefertigt habe und keine  
anderen als die angeführten Hilfen benutzt habe;

2. dass ich die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern  
und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die  
Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen  
innerhalb der Arbeit gekennzeichnet habe.
3. dass die eingereichten Abgabe-Exemplare in Papierform und im PDF-Format  
vollständig übereinstimmen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

**Konstanz, 18. Juni 2025**

.....  
(Julian Rapp)

## Zitat

*„Interrupts are a major feature of most embedded software. They are vaguely like functions that are called by hardware rather than software. The distinction sounds trivial but it makes them much harder to handle [...].“*

(DAVIES)

# Abstract

Die vorliegende wissenschaftliche Arbeit behandelt die Konzeption und Umsetzung einer interruptgesteuerten Benutzerschnittstelle zur Überwachung und Manipulation von Registern und Speicherzellen auf einem Low-Power-Mikrocontroller der MSP430-Familie von Texas Instruments. Im Zentrum der Entwicklung steht das sogenannte Observer-Modul, das eine statusorientierte, nicht-blockierende Abarbeitung eingehender Steuerbefehle erlaubt und damit die Grundlage für eine interaktive Systemanalyse und gezieltes Debugging im Echtzeitbetrieb schafft.

Im Rahmen der Entwicklung wurde unter anderem ein zustandsbasierter Automat, eine UART-Kommunikationsschnittstelle sowie ein konfigurierbaren Timer-Interrupt realisiert, um eine zeichenweise Verarbeitung von Befehlen bei gleichzeitiger Entkopplung vom Hauptprogramm zu ermöglichen. Die Implementierung wurde durch eine detaillierte Analyse der Systemreaktivität mittels GPIO-Signalisierung und Oszilloskop gestützt, wodurch Aussagen über die Echtzeiteigenschaften einzelner Funktionen getroffen werden konnten.

Ein besonderes Augenmerk lag auf der prototypischen Integration softwarebasierter Breakpoints, welche tiefergehende Eingriffe in den Programmablauf ermöglichen sollen. Trotz des hohen Komplexitätsgrades konnte durch die Konzeptionierung eine solide Basis geschaffen werden, die weiterführende Forschung und Entwicklung im Bereich eingebetteter Debugging-Technologien unterstützt.

Die fertige Erweiterung wurde im Rahmen des Praktikums Mikroprozessorsysteme erfolgreich unter Live-Bedingungen getestet und demonstriert die praktische Relevanz und Flexibilität der entwickelten Lösung.

# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>I</b>
<b>Abkürzungsverzeichnis und Glossar</b>	<b>III</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Das Ziel dieser Arbeit . . . . .	1
1.2. Die Umgebung, in der die Arbeit entstand . . . . .	1
1.3. Der Inhalt dieser Arbeit . . . . .	2
1.4. Viele Informationen, wenige Quellen ... . . . .	2
<b>2. Technische Hintergründe</b>	<b>3</b>
2.1. Timer und Interrupt Service Routinen (ISR) . . . . .	5
2.1.1. Timer Zählweisen . . . . .	7
2.1.2. Capture-Mode . . . . .	9
2.1.3. Compare-Mode . . . . .	10
2.1.4. Einstellungen der Capture and Compare Register . . . . .	12
2.1.5. Timer Control-Register . . . . .	15
2.1.6. Zusammenfassung der Einsatzmöglichkeiten . . . . .	17
2.2. Enhanced Universal Serial Communication Interface (eUSCI) . . . . .	19
2.2.1. Die eUSCI-Architektur . . . . .	20
2.2.2. Einordnung vorhandener Kommunikationsschnittstellen . . . . .	21
2.2.3. eUSCI_A Modul: UART-Modus (Asynchrone Kommunikation) . . . . .	23
2.2.4. eUSCI-Konfiguration . . . . .	29
2.2.5. Zusammenfassung . . . . .	31
<b>3. Planung und praktische Umsetzung</b>	<b>33</b>
3.1. Konzeptualisierung Observer-Modul . . . . .	34
3.1.1. Zustandsautomat und Funktionsumfang . . . . .	35
3.1.2. Initialisierung . . . . .	36
3.1.3. Timer ISR . . . . .	38
3.1.4. UART ISR . . . . .	40
3.2. Zustandsmaschine und Hilfsstrukturen . . . . .	43
3.2.1. Die Zustandsmaschine . . . . .	43
3.2.2. Zeichenweise Ausgabe . . . . .	46

---

3.3. Interruptgesteuertes Lesen und Schreiben . . . . .	47
3.3.1. Grundkonzept des statusorientierten, interruptgesteuerten Betriebs . . . . .	47
3.3.2. Rolle der Interrupt Service Routinen . . . . .	48
3.3.3. Implementierung der Lese- und Schreibfunktion . . . . .	49
3.4. Fehlerbehandlung . . . . .	54
3.5. Debugging-Methoden: Hardware- vs. Software-Breakpoints . . . . .	58
3.5.1. Der MSP-FET Download Adapter im Detail . . . . .	60
3.5.2. Konzeptionierung von Software-Breakpoints . . . . .	63
<b>4. Fazit und kritische Bewertung</b>	<b>69</b>
4.1. Das Ergebnis . . . . .	69
4.1.1. Untersuchung der Laufzeit unter Realbedingungen . . . . .	70
4.1.2. Fazit zu Software Breakpoints . . . . .	73
4.2. Die Bewertung . . . . .	73
4.3. Ein Ausblick . . . . .	75
<b>Literatur</b>	<b>VII</b>
<b>Abbildungsverzeichnis</b>	<b>X</b>
<b>Tabellenverzeichnis</b>	<b>XI</b>
<b>Verzeichnis der Listings</b>	<b>XII</b>
<b>A. Anhang</b>	<b>i</b>
A.1. Verwendete Hilfsmittel . . . . .	i
A.1.1. Erklärung zur Nutzung von KI-Sprachmodellen zur sprachlichen Überarbeitung . . . . .	i
A.1.2. Erklärung zur Erstellung von Diagrammen . . . . .	ii

## Abkürzungsverzeichnis und Glossar

16-Bit-RISC-MCU	Mikrocontroller mit 16-Bit-Registerbreite und reduzierter Befehlssatzarchitektur (Reduced Instruction Set Computer). [14]
Absolute .....	Operand befindet sich an einer feste Speicheradresse.
ACLK .....	Auxiliary Clock
Adressierungsmodus	Der Adressierungsmodus bezeichnet die Art und Weise, wie Operanden innerhalb des Befehlswortes adressiert werden; unterstützt werden u. a. Register-, symbolische, absolute und indirekte Adressierung. [11, S. 97, Kap. 4.4]
Adresskomparator	Adresskomparatoren vergleichen Speicheradressen zweier Operanden, um deren Gleichheit oder relative Ordnung (größer/kleiner) festzustellen, typischerweise zur Steuerung von Sprüngen oder Zugriffsentscheidungen in Programmen.
alphanumerisch ....	Zeichenkombination aus Buchstaben (lateinisches Alphabet) und Ziffern (0–9).
Auxiliary Clock ...	Niederfrequente Taktquelle in Mikrocontroller-Systemen, die typischerweise von einem Quarzoszillator gespeist wird und für energiesparende Betriebsmodi verwendet wird.
Baudrate .....	Bezeichnet die Anzahl der Signaländerungen (Symbole) pro Sekunde in einem Übertragungskanal. Sie bestimmt somit die Übertragungsgeschwindigkeit, ist aber nicht zwingend identisch mit der Bitrate, da pro Symbol mehrere Bits codiert sein können. [21]
BITCLK .....	Bit-Clock
Breakpoint .....	Bezeichnet in der Softwareentwicklung eine vom Entwickler bewusst gesetzte Unterbrechung im Programmablauf, die typischerweise zur Laufzeit-Debugging-Zwecken verwendet wird. Beim Erreichen dieses Punkts wird die Ausführung des Programms angehalten, sodass der aktuelle Zustand (z. B. Variablen, Stack, Speicher) analysiert werden kann.
Clock-Source .....	Eine Referenz auf ein periodisches Zeitsignal um zeitliche Abläufe zu synchronisieren; typischerweise in Form von Quarzoszillatoren oder externen Taktsignalen.

---

Compiler-Intrinsic	Compiler-spezifische, vordefinierte Funktionen als optimierter Assemblercode.
CPU	Central Processing Unit
Debugger	Ein Werkzeug zur schrittweisen Ausführung und Analyse von Programmen. Es erlaubt das Setzen von Haltepunkten, das überprüfen von Speicherinhalten und das Nachvollziehen von Kontrollflüssen zur Fehlersuche und -behebung.
Dictionary	Datenstruktur, die Schlüssel-Wert-Paare speichert und schnellen Zugriff auf Werte über ihre eindeutigen Schlüssel erlaubt.
EEM	Embedded Emulation Module
enhanced Universal Serial Communication Interface	Serielle Schnittstelle in Mikrocontrollern von Texas Instruments, die verschiedene Kommunikationsprotokolle (z. B. UART, SPI, I <sup>2</sup> C) unterstützt.
eUSCI	enhanced Universal Serial Communication Interface
eUSCI_A	enhanced Universal Serial Communication Interface Type A
eUSCI_B	enhanced Universal Serial Communication Interface Type B
extended Word	Synonym für 4 Byte große Datenbreite.
Flag	Ein Flag bezeichnet in der Informatik ein einzelnes Bit oder eine boolesche Variable, die einen binären Zustand (z. B. „wahr/falsch“ oder „aktiv/inaktiv“) repräsentiert und häufig zur Steuerung von Programmabläufen oder zur Statusanzeige verwendet wird.
FRAM	Ferroelectric Random Access Memory
Funktionsprototyp	Ein Funktionsprototyp legt Signatur und Rückgabotyp einer Funktion fest, ohne deren Implementierung bereitzustellen.
GEL	General Extension Language
General-Purpose-Register	Frei verwendbare Arbeitsregister für allgemeine Operationen.
GPIO	General Purpose Input/Output
Halbduplex	Datenübertragung zu einem Zeitpunkt nur in eine Richtung möglich. [18]
Handler	Ein Handler ist in der Informatik eine Routine oder Funktion, die auf ein bestimmtes Ereignis oder eine Bedingung reagiert, z. B. ein Ereignis- oder Interrupt-Handler.
High-Pegel	Bezeichnet in der Digitaltechnik einen Spannungszustand, der über einer definierten Schaltschwelle liegt. [7]
I <sup>2</sup> C	Inter-Integrated Circuit



---

IDE .....	Integrated Development Environment
Immediate .....	Operand ist direkt in der Instruktion enthalten – also ein fester Wert, nicht aus dem Speicher.
Indirect .....	Operand ist nicht direkt gegeben, sondern steht an der Adresse, die ein Register enthält.
Interface .....	Ein Interface bezeichnet in der Informatik eine definierte Schnittstelle, über die Komponenten eines Softwaresystems miteinander kommunizieren können, ohne deren interne Implementierung zu kennen.
interruptgesteuert .	Ein Mechanismus zur ereignisorientierten Unterbrechung des normalen Programmablaufs.
ISR .....	Interrupt Service Routine
JTAG .....	Joint Test Action Group
LIN .....	Local Interconnected Network
Local Interconnected Network	Serieller Kommunikationsstandard, zur kostengünstigen Vernetzung von Mikrocontrollern in Fahrzeugen, insbesondere bei weniger zeitkritischen Steuergeräten. [20]
Low-Pegel .....	Bezeichnet in der Digitaltechnik einen Spannungszustand, der unter einer definierten Schaltschwelle liegt. [7]
Low-Power-Mikrocontroller	Ein Mikrocontroller, der für energieeffiziente Anwendungen optimiert ist. Typischerweise eingesetzt in batteriebetriebenen Embedded-Systemen.
LP-MCU .....	Low-Power-Mikrocontroller
LPM .....	Low Power Mode
LSB .....	Least Significant Bit
Makro .....	Ein Makro ist eine Anweisung oder ein Symbol, das durch eine andere Zeichenfolge ersetzt wird, typischerweise zur Automatisierung wiederkehrender Codeabschnitte oder Befehle.
MC .....	Mode Control
Megahertz .....	Maßeinheit für die Frequenz und entspricht einer Million Schwingungen pro Sekunde ( $1 \text{ MHz} = 10^6$ Rechenschritte). [17]
MIPS .....	Mikroprozessorsysteme
Modul .....	Eine funktionale Einheit innerhalb eines größeren Systems, die separat entwickelt und gewartet werden kann.
MPU .....	Memory Protection Unit
MS/s .....	Mega-Samples pro Sekunde

---

MSB .....	Most Significant Bit
MSP-FET .....	MSP430 Flash Emulation Tool
Non-Return-to-Zero	Binäres Leitungscodierungsverfahren, bei dem der Signalpegel während eines Bitintervalls konstant bleibt und nicht zwischen den Bits auf einen Nullpegel zurückkehrt. [9]
NRZ .....	non-return to zero
Opcode .....	Auch op code oder operation code, ist eine meist in hexadezimaler Schreibweise angegebene Zahl, die die Nummer eines Maschinenbefehls für einen bestimmten Prozessortyp angibt.
Oversampling .....	Oversampling bezeichnet das Verfahren, bei dem ein Signal mit einer höheren Abtastrate als der Abtastgrenze (Nyquist-Rate) digitalisiert wird, um Rauscheinflüsse zu reduzieren und die Signalqualität zu verbessern. [4]
PC .....	Personal Computer
PC .....	Program Counter
Plug-and-Play .....	Automatische Erkennung und Integration von Komponenten in ein System ohne manuelle Konfiguration. [19]
Prescaler .....	Vorschaltglied in elektronischen Zählschaltungen oder Timern, welches die Frequenz eines Eingangssignals durch einen festen Faktor reduziert, um eine nachfolgende Verarbeitung mit geringerer Taktrate zu ermöglichen.
Program Counter ..	Ein Register, das die Speicheradresse des derzeitigen Befehls enthält.
Pulsweitenmodulation	Ein Verfahren zur Steuerung der Leistungszufuhr, bei dem die mittlere Ausgangsleistung durch Variieren des Abtastverhältnisses eines Rechtecksignals reguliert wird. [4]
PWM .....	Pulsweitenmodulation
RAM .....	Random Access Memory
Register .....	Speicherzellen, die flüchtig sind und ihre Inhalte beim Ausschalten verlieren. [5]
ROM .....	Read Only Memory
RxD .....	Receive Data
sampling clock .....	Bestimmt den Zeitpunkt der Abtastung eingehender Bits; sie wird üblicherweise aus einer übergeordneten Taktquelle (z. B. SMCLK) abgeleitet und beeinflusst maßgeblich die Genauigkeit der Datenübertragung. [4]

---

SBW .....	Spy-Bi-Wire
Serielle Kommunikation	Die Übertragung von Datenbitfolgen über eine einzelne Leitung, wobei die Bits nacheinander (also „seriell“) gesendet werden. Sie wird häufig bei einfachen oder ressourcenschonenden Datenverbindungen eingesetzt, z. B. über UART, SPI oder I <sup>2</sup> C.
SMCLK .....	Sub-Main Clock
SP .....	Stack Pointer
SP .....	Stop-Bit
SPI .....	Serial Peripheral Interface
Spy-Bi-Wire .....	Zweidraht-Variante des JTAG-Protokolls, die Pin-Anzahl am Target reduziert und besonders für platzkritische Anwendungen von Vorteil ist.
SR .....	Status Register
SRAM .....	Static Random Access Memory
ST .....	Start-Bit
Stack Pointer .....	Ein Register, das die Speicheradresse des letzten oder ersten Datenelements im Stack speichert.
statischer Arbeitsspeicher	Schnellster, flüchtiger Speicher mit geringer Kapazität, bestehend aus Flip-Flops welcher meist direkt in der <i>CPU</i> mit eingebaut ist. <a href="#">[12]</a>
Statusregister .....	Register für eine Reihe von Flags, die von der arithmetisch-logischen Einheit in Abhängigkeit der zuletzt durchgeführten Rechenoperation gesetzt werden.
Sub-Main Clock ...	Taktgesteuertes Signal, das typischerweise für Peripheriegeräte verwendet wird und sich aus einer frei wählbaren Taktquelle ableiten lässt.
Target .....	Das Zielobjekt oder die Zielumgebung, auf das bzw. in die eine Operation, ein Befehl oder eine Ausführung abzielt – z. B. bei Compilern die Zielplattform, bei Build-Systemen das zu erstellende Artefakt.
Tool .....	Ein Softwarewerkzeug, das spezifische Aufgaben oder Funktionen innerhalb eines größeren Systems oder Workflows erfüllt.
Trade-off .....	Abwägung zwischen zwei konkurrierenden Zielen, Konzepten, oder ähnlichem, bei der die Verbesserung des einen mit der Verschlechterung des anderen einhergeht.

---

TxD .....	Transmit Data
UART .....	Universal Asynchronous Receiver Transmitter
Vollduplex .....	Gleichzeitige Datenübertragung in beide Richtungen. <a href="#">[18]</a>
Word .....	Synonym für 2 Byte große Datenbreite.
Zustandsautomat ..	Auch endlicher Automat, ist ein mathematisches Modell zur Beschreibung von Systemen mit endlich vielen Zuständen, das Übergänge zwischen diesen in Abhängigkeit von Eingaben definiert.

# 1. Einleitung

## 1.1. Das Ziel dieser Arbeit

Diese Bachelor-Thesis befasst sich mit der Entwicklung einer *interruptgesteuerten*<sup>1</sup> Benutzerschnittstelle auf einem *Low-Power-Mikrocontroller*<sup>2</sup>, zur Überwachung und Manipulation von *Registern*<sup>3</sup> und Speicherzellen in *RAM* und *FRAM*.

Im Zuge des Arbeitsauftrags entwickelt diese Arbeit ein unabhängiges *Modul*<sup>4</sup>, das sich bei Bedarf aktivieren oder deaktivieren lässt.

## 1.2. Die Umgebung, in der die Arbeit entstand

Die Entwicklung der Software geschieht in Absprache mit Herrn Prof. Dr. Irenäus Schoppa, der ein zusätzliches *Tool*<sup>5</sup> zur tiefgreifenden Analyse unter Realbedingungen von Register- und Speicherinhalten für Studenten im Praktikum von Mikroprozessorsysteme benötigt.

Als Entwicklungsbasis kommt der in *MIPS* herangezogene MSP430FR5729 von Texas Instruments zum Einsatz, welcher bereits ein ausgereifter und etablierter *LP-MCU* ist. Der Fokus dieser Arbeit liegt auf den für die Implementierung wesentlichen Aspekten. Zahlreiche dem Prozessor zugrundeliegende Technologien werden aus diesem Grund bewusst ausgeklammert. [4]

---

<sup>1</sup>Ein Mechanismus zur ereignisorientierten Unterbrechung des normalen Programmablaufs.

<sup>2</sup>Ein Mikrocontroller, der für energieeffiziente Anwendungen optimiert ist. Typischerweise eingesetzt in batteriebetriebenen Embedded-Systemen.

<sup>3</sup>Speicherzellen, die flüchtig sind und ihre Inhalte beim Ausschalten verlieren. [5]

<sup>4</sup>Eine funktionale Einheit innerhalb eines größeren Systems, die separat entwickelt und gewartet werden kann.

<sup>5</sup>Ein Softwarewerkzeug, das spezifische Aufgaben oder Funktionen innerhalb eines größeren Systems oder Workflows erfüllt.

## 1.3. Der Inhalt dieser Arbeit

**Grundlagen, Technologien und Evaluation:** Kapitel 2 erarbeitet die für die Planung und Umsetzung verwendeten Grundlagen und Technologien. Das Kapitel erklärt die grundlegenden Eigenschaften und den Aufbau des Softwareentwicklungsprozesses auf einem LP-MCU und referenziert die benötigten Dokumentationen.

**Konzeptionierung der Benutzerschnittstelle:** Dieses Kapitel umfasst die Dokumentation der gesamten Planungsphase des Observer-Moduls. Hier entsteht eine Übersicht über die Lösungsfindung, und die zur Planung erforderlichen Dokumente und Diagramme.

**Die Entwicklung des Observers:** Im folgenden steht die Dokumentation der tatsächlichen Programmierung des Software-Moduls im Vordergrund. Dieser Teil klärt die Implementierung und arbeitet den Entwicklungsverlauf anhand von Beispielen schrittweise ab.

**Fazit und kritische Bewertung:** Das Fazit fasst die gemachten Erfahrungen sowie die Ergebnisse der Planung und Entwicklung abschließend zusammen und bewertet sie kritisch. Zusätzlich bietet das Kapitel einen kurzen Ausblick auf Erweiterungsmöglichkeiten und potenzielle Optimierungsschritte.

## 1.4. Viele Informationen, wenige Quellen . . .

Grundsätzlich ist es einfach geeignete Quellen zu den Themen rund um den MSP430 und dessen integrierten Technologien zu finden, da – wie bereits erwähnt – der MSP430 weitestgehend etabliert ist. Der Hersteller – Texas Instruments – stellt dazu alle wesentlichen Informationen zur Softwareentwicklung bereit, was die Notwendigkeit vieler weiterer Quellen hinfällig macht.

## 2. Technische Hintergründe

Die gewählte Entwicklungsplattform ist entscheidend für die darauf folgende Implementierung eingebetteter Software. Im Rahmen dieser Arbeit dient der MSP430FR5729 von Texas Instruments als zentrale Hardwarekomponente. Dessen Technische Daten und Funktionalitäten werden in den folgenden Abschnitten näher betrachtet.

Der MSP430FR5729 ist ein Low-Power-Mikrocontroller *16-Bit-RISC-MCU*<sup>6</sup> von Texas Instruments mit einer Maximalen Taktfrequenz von Acht *Megahertz*<sup>7</sup>. Eingebaute Low-Power-Modi (*LPMs*) ermöglichen u. a. niedrigere Taktfrequenzen und das Deaktivieren von Oszillatoren, wodurch er sich besonders gut für energieeffiziente Anwendungen im Bereich eingebetteter Systeme eignet. Eine Auflistung aller Modi sind in Abbildung 2.1 zu sehen.

[8, 11, S. 43, Kap. 6.3, S. 35, Kap. 1.4 & S. 37, Kap. 1.4.1]

SCG1 <sup>(1)</sup>	SCG0	OSCOFF <sup>(1)</sup>	CPUOFF <sup>(1)</sup>	Mode	CPU and Clocks Status <sup>(2)</sup>
0	0	0	0	Active	CPU, MCLK are active. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK, MCLK, or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0).
0	0	0	1	LPM0	CPU, MCLK are disabled. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0).
0	1	0	1	LPM1	CPU, MCLK are disabled. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0).
1	0	0	1	LPM2	CPU, MCLK are disabled. ACLK is active. SMCLK is disabled. DCO is enabled if sources ACLK.
1	1	0	1	LPM3	CPU, MCLK are disabled. ACLK is active. SMCLK is disabled.
1	1	1	1	LPM4	CPU and all clocks are disabled.
1	1	1	1	LPM3.5	When PMMREGOFF = 1, regulator is disabled. No memory retention. In this mode, RTC operation is possible when configured properly. See the <i>RTC</i> module for further details.
1	1	1	1	LPM4.5	When PMMREGOFF = 1, regulator is disabled. No memory retention. In this mode, all clock sources are disabled; that is, no RTC operation is possible.

<sup>(1)</sup> This bit is automatically reset when exiting low-power modes. See Section 1.4.2 for details.

<sup>(2)</sup> The low-power modes and, hence, the system clocks can be affected by the clock request system. See the [Clock System](#) chapter for details.

Abb. 2.1.: Operating Modes [11, S. 37, Kap. 1.4, Tab. 1-2]

<sup>6</sup>Mikrocontroller mit 16-Bit-Registerbreite und reduzierter Befehlssatzarchitektur (Reduced Instruction Set Computer). [14]

<sup>7</sup>Maßeinheit für die Frequenz und entspricht einer Million Schwingungen pro Sekunde (1 MHz = 10<sup>6</sup> Rechenschritte). [17]

Die Versorgungsspannung beträgt 2 bis 3,6 Volt wobei ebenfalls verschiedene Low-Power-Modi verwendet werden können, um den Stromverbrauch zunehmend zu minimieren. Diese beeinflussen den späteren Umgang mit Timer-Interrupts, weil sie den Energieverbrauch im Wartezustand beeinflussen. [8, S. 26, Kap. 5.20]

Der Mikrocontroller besitzt 16 Kilobyte an nicht-flüchtigen FRAM, sowie ein Kilobyte *statischen Arbeitsspeicher*<sup>8</sup> (*SRAM*). [8, S. 1, Kap. 1.1]

Die daran anhängende Memory Protection Unit (*MPU*) – falls aktiviert – verhindert dabei versehentliche Schreibzugriffe auf FRAM Speicherzellen.

[11, S. 252, Kap. 6.1]

Des Weiteren besitzt der Chip Fünf Interne 16-Bit Timer mit jeweils Sieben **Capture and Compare** Registerblöcken. Diese internen Timer stellen eine zentrale Komponente für die Realisierung präziser Zeitgesteuerter Funktionen und die Generierung von Interrupts dar, welche im nachfolgenden Kapitel 2.1 tiefergreifender erläutert werden.

Zur externen Kommunikation sind Protokolle wie *UART*, *I<sup>2</sup>C* und *SPI* integriert, welche mit 32 programmierbaren *GPIO*-Pins angeschlossen werden können. Kommunikationsschnittstellen sind für die Interaktion mit der Außenwelt und Peripheriegeräten von hoher Bedeutung. Eine detailliertere Ausarbeitung des *enhanced Universal Serial Communication Interface*<sup>9</sup> (*eUSCI*) folgt in Kapitel 2.2.

[8, S. 1, Kap. 1.1]

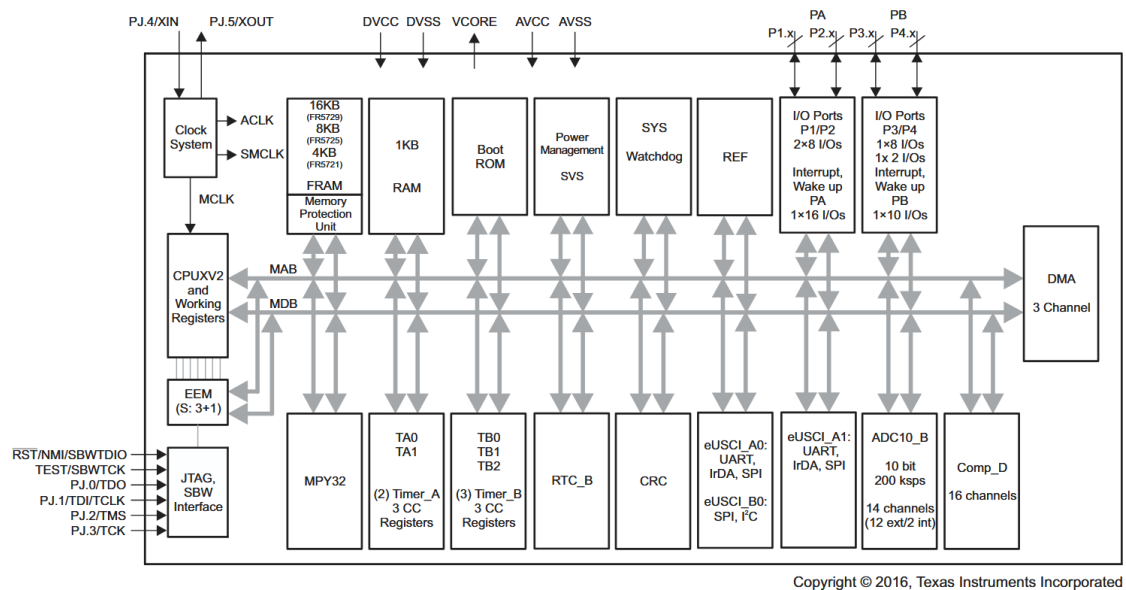
---

<sup>8</sup>Schnellster, flüchtiger Speicher mit geringer Kapazität, bestehend aus Flip-Flops welcher meist direkt in der *CPU* mit eingebaut ist. [12]

<sup>9</sup>Serielle Schnittstelle in Mikrocontrollern von Texas Instruments, die verschiedene Kommunikationsprotokolle (z. B. *UART*, *SPI*, *I<sup>2</sup>C*) unterstützt.



Abbildung 2.2 zeigt ein vollständiges Block-Diagramm des Mikrocontrollers, welches noch einige weitere Eigenschaften, Funktionen und Subsysteme auflistet.<sup>10</sup>



**Abb. 2.2.:** Block Diagramm MSP430FR5729  
Mikrocontroller [8, S. 2, Kap. 1.4]

## 2.1. Timer und Interrupt Service Routinen (ISR)

Timer und Interrupt Service Routinen (*ISR's*) stellen einen fundamentalen Baustein moderner eingebetteter Systeme dar. Sie ermöglichen präzise, zeitgesteuerte Funktionen sowie das Reagieren auf externe Ereignisse. Womit die Realisierungen komplexer, Echtzeitsysteme möglich wird. Im Folgenden wird die Timer-Architektur des MSP430FR5729 und die zugehörigen ISR-Mechanismen detailliert betrachtet.

Der MSP430FR5729 verfügt über insgesamt fünf 16-Bit-Timer, wovon zwei zu Typ A und drei zu Typ B gehören. Timer haben vielseitige Einsatzgebiete und ermöglichen verschiedenste Zeitsteuerungsfunktionen. Die Unterscheidung in zwei Typen lässt dabei eine größere Vielfalt an spezifischen Konfigurationsmöglichkeiten zu.

<sup>10</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

Beide Timer-Typen verfügen über einen gemeinsamen 16-Bit-Zähler sowie sieben Capture/Compare-Register. Die korrekte Konfiguration dieser Register ist ausschlaggebend für die Implementierung verschiedenster Funktionen. Eine der beiden übergeordneten Eigenschaften, die Capture-Funktionalität, dient dazu, den aktuellen Zählerwert bei einem externen oder internen Ereignis präzise zu erfassen. Dies ist beispielsweise nützlich für die Messung von Pulsweiten oder Frequenzen. Die Compare-Funktionalität hingegen erlaubt den Vergleich des aktuellen Zählerstandes mit einem in den Compare-Registern hinterlegten Wert. Bei einer Übereinstimmung kann eine konfigurierbare Aktion ausgelöst werden, wie beispielsweise das Setzen oder Rücksetzen eines Ausgangspins oder das Generieren eines Interrupts. Ausführlicher beleuchtet ab Kapitel 2.1.2. Die vielseitigen Einstellungsmöglichkeiten dieser und weiterer Register erlauben die Realisierung komplexer Zeitgesteuerter Aufgaben.

[11, 4, S. 333, Kap. 11 & S. 355, Kap. 12, S. 287, Kap. 8.3 & S. 194, Kap. 6.8.2]

Die Timer des Typs B weisen im Vergleich zu dem Timer des Typs A, erweiterte Konfigurationsmöglichkeiten auf. Darunter fällt die Konfigurierbarkeit der Timer-Länge auf 8, 10, 12 oder 16 Bit, was eine flexible Anpassung der Zählauflösung und der Überlaufperiode für unterschiedliche Auflösungen ermöglicht. Zusätzlich sind alle Capture/Compare-Blöcke doppelt gepuffert. Diese doppelte Pufferung erlaubt das Laden neuer Vergleichswerte, während eines aktiven Zählzyklus. Unerwünschte Effekte oder Inkonsistenzen in den Ausgangssignalen können dadurch vermieden werden. Ein weiterer Unterschied besteht darin, dass die Capture/Compare-Eingänge asynchron zueinander sind. Somit können sie unabhängig zum internen Takt des Timers operieren, was in bestimmten Szenarien die Erfassung externer Ereignisse erleichtert. [11, 4, S. 356, Kap. 12.1.1, S. 353, Kap. 8.9]

Für die präzise Steuerung und Ereignisbehandlung bieten die Timer verschiedene Betriebsarten, die im Folgenden näher erläutert werden.

### 2.1.1. Timer Zählweisen

Der Zählmodus, bestimmt die interne Zählweise des Timers. Die Timer unterstützen typischerweise mehrere Varianten, um unterschiedlichen Anforderungen gerecht zu werden. [4, S. 291, Kap. 8.3.1]

- **Up Mode:** Im Up Mode (Additive Zählweise, Vgl. Abbildung 2.3) beginnt der Zähler bei Null und inkrementiert seinen Wert mit jedem Taktimpuls der gewählten *Clock-Source*<sup>11</sup>. Er erreicht einen vordefinierten Maximalwert, der im Compare-Register gespeichert ist, und beginnt dann wieder von Null an zu zählen. Ein überlauf-Interrupt wird generiert, sobald der Zähler den Wert von CCR0 erreicht. Dieser Modus eignet sich ideal für die Erzeugung periodischer Ereignisse oder die Messung von Zeitintervallen bis zu einem bestimmten Grenzwert. Beispielsweise kann durch die Wahl einer geeigneten Clock-Source und eines passenden Wertes im Compare-Register eine präzise Zeitbasis für periodische Aufgaben geschaffen werden.

[11, 4, S. 337, Kap. 11.2.3.1 & S. 359, Kap. 12.2.3.1, S. 330, Kap. 8.6]

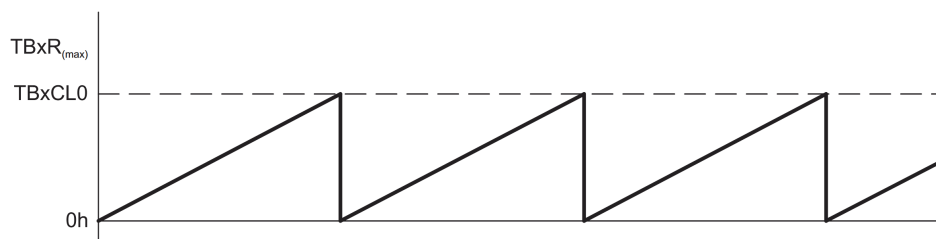


Abb. 2.3.: Up Mode [11, S. 359, Abb. 12.2]

<sup>11</sup>Eine Referenz auf ein periodisches Zeitsignal um zeitliche Abläufe zu synchronisieren; typischerweise in Form von Quarzoszillatoren oder externen Taktsignalen.

- **Continuous Mode:** Der Continuous Mode (Vgl. Abbildung 2.4) lässt den Zähler von Null bis zum maximal möglichen Wert (FFFFh für 16-Bit-Timer) zählen und anschließend wieder bei Null beginnen. Ein überlauf-Interrupt wird generiert, wenn der Zähler vom Wert von FFFFh auf 0 überläuft. [11, S. 338, Kap. 11.2.3.2 & S. 360, Kap. 12.2.3.2] Dieser Modus ist besonders nützlich, wenn längere, voneinander unabhängige Zeitintervalle zu messen sind oder eine freilaufende Zeitbasis benötigt wird, um Ereignisse in Bezug auf den Zählerstand, ohne einen periodischen Neustart durch das Compare-Register, zu erfassen. [11, 4, S. 338, Kap. 11.2.3.3 & S. 360, Kap. 12.2.3.3, S. 318, Kap. 8.5]

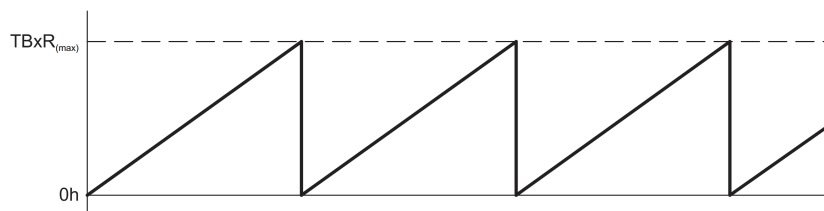


Abb. 2.4.: Continuous Mode [11, S. 360, Abb. 12.4]

- **Up/Down Mode:** Der Up/Down Mode (Auf-/Abwärtszählmodus, Vgl. Abbildung 2.5) kombiniert das Auf- und Abzählen. Der Zähler beginnt bei Null, zählt zyklisch bis zum festgelegten Wert im Compare-Register und dann wieder bis Null herunter. Ein überlauf-Interrupt wird generiert, wenn der Zähler den Wert von CCR0 erreicht, und ein weiterer Interrupt (sofern aktiviert) kann beim Erreichen von Null gesetzt werden. [11, S. 339, Kap. 11.2.3.4 & S. 361, Kap. 12.2.3.4] Dieser Modus erzeugt eine symmetrische *Pulsweitenmodulation*<sup>12</sup> (*PWM*) und wird häufig in Anwendungen zur Motorsteuerung oder zur Erzeugung präziser analoger Ausgangssignale eingesetzt. [11, 4, S. 340, Kap. 11.2.3.5 & S. 362, Kap. 12.2.3.5, S. 349, Kap. 8.7]

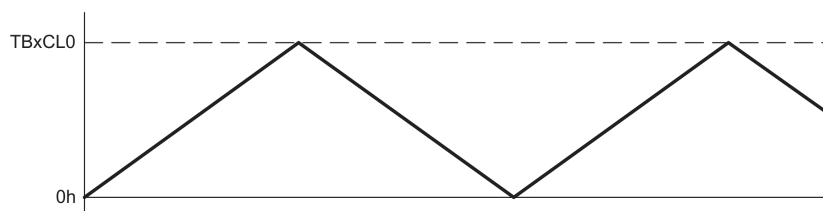


Abb. 2.5.: Up/Down Mode [11, S. 361, Abb. 12.7]

<sup>12</sup>Ein Verfahren zur Steuerung der Leistungszufuhr, bei dem die mittlere Ausgangsleistung durch Variieren des Abtastverhältnisses eines Rechtecksignals reguliert wird. [4]

Die Wahl eines geeigneten Modus hängt stark von der spezifischen Anwendung ab. Für einfache Zeitmessungen oder periodische Aufgaben ist der Up Mode oft ausreichend, während der Continuous Mode für längere Intervalle oder als Basis für komplexere Zeitsteuerungen dient. Der Up/Down Mode hingegen findet seine Anwendung primär in der Erzeugung von Steuersignalen.

Zur Ergänzung der drei Zählweisen des Timers, wird in den folgenden Kapiteln die genaue Rolle der bereits erwähnten Capture-/Compare-Modi erläutert.

### 2.1.2. Capture-Mode

Der Capture Mode ermöglicht es, den aktuellen Wert des Zählers präzise zu erfassen, wenn ein bestimmtes Ereignis an einem zugehörigen Eingangspin auftritt. Der erfasste Zählerwert wird in einem der Capture-Register (CCR0 bis CCR6) gespeichert. Dies ist besonders nützlich für die Messung von externen Signalen wie Pulsweiten, Frequenzen oder der Zeit zwischen zwei Ereignissen. Beispiele hierzu in Abbildung 2.6.

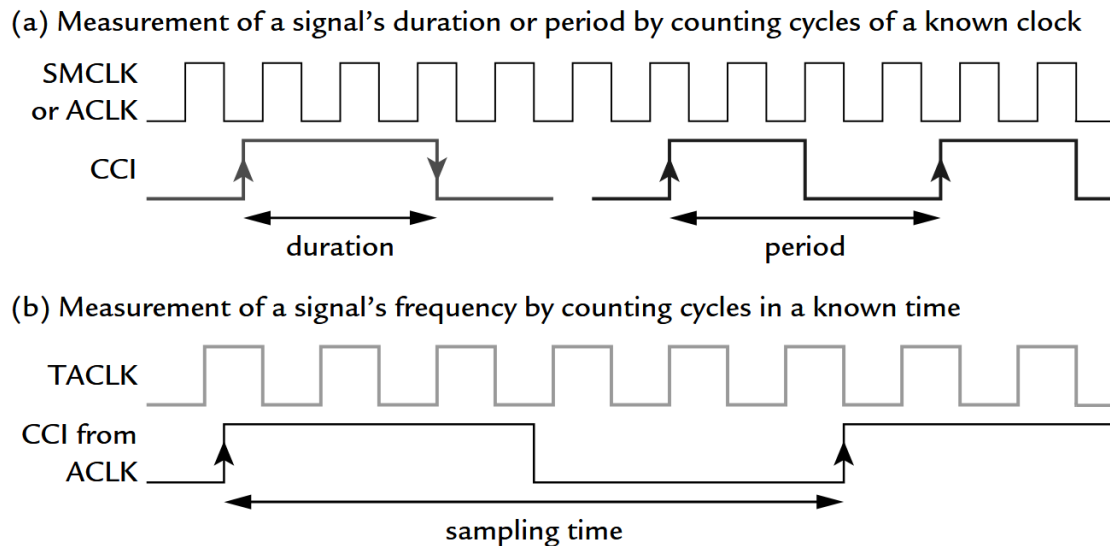


Abb. 2.6.: Capture Mode Einsatzbeispiele  
[4, S. 301, Abb. 8.7]

Die Timer des MSP430FR5729 unterstützen verschiedene Capture-Modi. Diese legen fest, bei welcher Art von Signaländerung die Erfassung des Zählerwertes erfolgt:

- **Capture on rising edge:** Sobald am zugehörigen Eingangspin eine steigende Flanke detektiert wird (übergang von Low nach High) wird in diesem Modus der aktuelle Zählerwert in das Capture-Register geschrieben.
- **Capture on falling edge:** Hier erfolgt die Erfassung des Zählerwertes am Eingangspin bei einer fallenden Flanke (übergang von High nach Low).
- **Capture on both edges:** Dieser Modus ermöglicht die Erfassung des Zählerwertes sowohl bei steigender als auch fallender Flanken. Dies ist besonders praktisch für die Messung von Signalperioden oder bei Relevanz beider Flanken eines Signals.

Sofern ein Interrupt im entsprechenden Capture-Register aktiviert ist, kann dieser auch Interrupts auslösen. In der zugehörigen ISR kann der erfasste Zählerwert aus dem Capture-Register gelesen und weiterverarbeitet werden. Mehrere Capture-Register innerhalb eines Timers ermöglichen die Erfassung und Auswertung mehrerer aufeinanderfolgender Ereignisse, ohne dass der vorherige Wert überschrieben wird.

Die Konfiguration des Capture Mode umfasst die Auswahl des auslösenden Ereignisses (Flanke) sowie ggf. die Aktivierung des Capture-Interrupts. Die erfassten Zeitstempel im Capture-Register erlauben präzise Messungen und die Analyse externer Signale in eingebetteten Systemen.

[11, 4, S. 340, Kap. 11.2.4.1 & S. 362, Kap. 12.2.4.1, S. 300, Kap. 8.4]

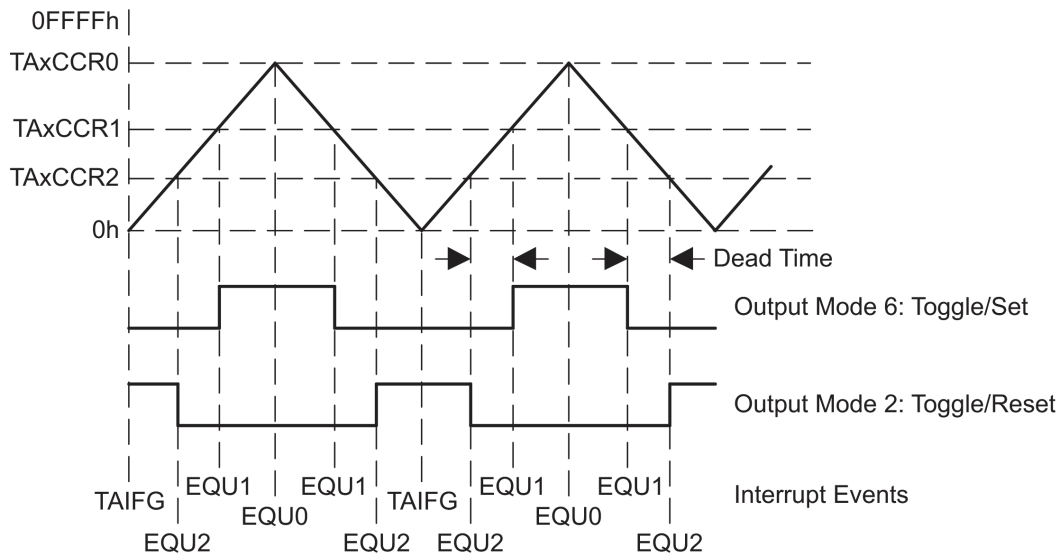
### 2.1.3. Compare-Mode

Der Compare Mode ermöglicht es, den aktuellen Wert des Zählers kontinuierlich mit den in den Compare-Registern CCR0 bis CCR7 hinterlegten Werten zu vergleichen. Wenn der Zählerstand mit dem Vergleichswert übereinstimmt, kann z. B. ein Interrupt ausgelöst oder ein Ausgangspin beeinflusst werden.

Die Compare-Modi bieten verschiedene Möglichkeiten, wie der Ausgangspin bei einer Übereinstimmung beeinflusst werden soll:

- **Set output on compare:** Bei einer Übereinstimmung des Zählerstandes mit dem Compare-Registerwert wird der zugehörige Ausgangspin auf High gesetzt.
- **Reset output on compare:** Hier wird der Ausgangspin bei Übereinstimmung auf Low gesetzt.
- **Toggle output on compare:** In diesem Modus ändert der Ausgangspin bei jeder Übereinstimmung seinen Zustand (von High nach Low oder von Low nach High).
- **Output High:** Der Ausgangspin wird permanent auf High gehalten.
- **Output Low:** Der Ausgangspin wird permanent auf Low gehalten.
- **Set/Reset:** In Kombination mit dem Compare-Register Null (CCR0) kann ein PWM-Signal erzeugt werden. Beispielsweise kann der Ausgang bei Erreichen des CCR0-Wertes gesetzt und bei Erreichen des CCRn-Wertes zurückgesetzt werden (oder umgekehrt), wobei CCRn die Pulsweite bestimmt.

Abbildung 2.7 zeigt eine mögliche Konfiguration im Zählmodus Up/Down mit zwei Compare-Registern (TAXCCR1 & TAXCCR2), eingestellt auf Toggle/Set und Toggle/Reset.



**Abb. 2.7.:** Ausgabereinheit im Up/Down-Modus  
[11, S. 340, Abb. 11-9]

Ähnlich wie beim Capture Mode ermöglicht ein Interrupt der CPU, auf präzise Zeitpunkte zu reagieren und entsprechende Aktionen auszuführen. Der Compare Mode ist somit ein vielseitiges Werkzeug zur Erzeugung von Steuersignalen, zur Implementierung von Zeitverzögerungen oder zur Synchronisation interner Operationen mit einer präzisen Zeitbasis.

[11, 4, S. 342, Kap. 11.2.4.2 & S. 364, Kap. 12.2.4.2, S. 352, Kap. 8.8]

Nachdem die verschiedenen Betriebsarten des Timers betrachtet wurden, ist es wichtig zu verstehen, wie die zugehörigen Register konfiguriert werden, um die gewünschte Funktionalität zu erzielen.

#### 2.1.4. Einstellungen der Capture and Compare Register

Die Konfiguration der zugehörigen Register bestimmt maßgeblich die Funktionalität der Capture- und Compare-Einheiten. Hierzu gehören die Aktivierung und Deaktivierung von Interrupts, die Auswahl des Ausgangsmodus (nur für Compare) sowie die Festlegung des auslösenden Ereignisses.

Für jedes Capture/Compare-Register kann individuell festgelegt werden, ob ein Interrupt ausgelöst werden soll, wenn ein entsprechendes Ereignis eintritt. Die Aktivierung erfolgt über spezifische **Interrupt-Enable-Bits** im jeweiligen Capture/Compare-Control-Register (TAXCCTLn oder TBxCCTLn).



Beispielsweise durch das Setzen des CCIE-Bits auf Eins oder Null, kann die Generierung eines Interrupts bei einem Capture- oder Compare-Ereignis aktiviert bzw. deaktiviert werden. Tabelle 2.1 fasst alle Register, beispielhaft anhand des Timers B, mit ihren Beschreibungen zusammen.

Bit	Feld	Typ	Reset	Beschreibung
15-14	CM	RW	0h	Capture mode
13-12	CCIS	RW	0h	Capture/compare input select. Über diese Bits wird das TBxCCRn Input-Signal ausgewählt.
11	SCS	RW	0h	Synchronize capture source. Dieses Bit entscheidet, ob das Capture-Input-Signal mit der Timer-Clock synchronisiert wird.
10-9	CLLD	RW	0h	Compare latch load. Auswählen des „compare latch load events“.
8	CAP	RW	0h	Capture-/Compare mode
7-5	OUTMOD	RW	0h	Output mode
4	CCIE	RW	0h	Capture/compare interrupt enable. Aktivieren der Interrupt-Anforderung des entsprechenden CCIFG-Flag <sup>13</sup> .
3	CCI	R	Undef	Capture/compare input. Das ausgewählte Eingangssignal kann über dieses Bit ausgelesen werden.
2	OUT	RW	0h	Output. Für Output-Mode „0“ kontrolliert dieses Bit den Ausgangsstatus.
1	COV	RW	0h	Capture overflow. Dieses Bit zeigt ein „capture overflow“ an. COV muss Softwareseitig zurückgesetzt werden.
0	CCIFG	RW	0h	Capture/compare interrupt flag

**Tab. 2.1.:** Registerbeschreibung – Capture-/Compare Register Timer B  
[11, S. 375, Tab. 12-8]

<sup>13</sup>Ein Flag bezeichnet in der Informatik ein einzelnes Bit oder eine boolesche Variable, die einen binären Zustand (z. B. „wahr/falsch“ oder „aktiv/inaktiv“) repräsentiert und häufig zur Steuerung von Programmabläufen oder zur Statusanzeige verwendet wird.

Wie in Kapitel 2.1.3 zum Compare-Modus erläutert, legen die Output-Mode-Bits (OUTMOD) fest, wie der zugehörige Ausgangspin bei einer Übereinstimmung des Zählerstandes mit dem Compare-Registerwert seinen Zustand verändert. Die Auswahl des passenden Output-Modus ist entscheidend für die Erzeugung der gewünschten Ausgangssignale, wie beispielsweise bei der Pulsweitenmodulation.

Die Auswahl des auslösenden Ereignisses für eine Capture- oder Compare-Operation wird ebenfalls über Bits im `TAxCTLn`- oder `TBxCTLn`-Register gesteuert. Für den Capture-Modus wird hier beispielsweise mit dem `CM`-Bit festgelegt, ob die Erfassung bei einer steigenden, fallenden oder bei beiden Flanken des Eingangssignals erfolgen soll.

Im Compare-Modus legt diese Einstellung fest, unter welchen Bedingungen die Vergleichsoperation erfolgreich ist und welche Aktion (Interrupt, Ausgangssignaländerung) dabei erfolgt. Dies kann beispielsweise ein reiner Vergleich oder auch ein Vergleich in Kombination mit dem Überlauf des Zählers im Up-Modus sein.

[11, 4, S. 351, Kap. 11.3.3 & S. 375, Kap. 12.3.3, S. 292, Kap. 8.3.2]

Die sorgfältige Konfiguration dieser Einstellungen in den Capture/Compare-Registern gewährleistet die Anpassung des Timers an die jeweiligen Anforderungen der Applikation.

Ein weiterer fundamentaler Aspekt der Timer-Konfiguration ist u. a. die Wahl der Taktquelle, welche die Zeitbasis für den Zähler und somit für alle zeitgesteuerten Operationen des Timers bestimmt.<sup>14</sup>

---

<sup>14</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

### 2.1.5. Timer Control-Register

Die Timer des MSP430FR5729 verwenden verschiedene interne Taktquellen, die jeweils unterschiedliche Eigenschaften aufweisen. Die primären Taktquellen sind *Auxiliary Clock*<sup>15</sup> (*ACLK*) und *Sub-Main Clock*<sup>16</sup> (*SMCLK*). Auch externe Taktquellen können zur Taktung des Timers herangezogen werden wie z. B. das *TACLK/TBCLK*-Register oder der *INCLK*-Pin.

[11, 4, S. 71, Kap. 3.1, S. 163, Kap. 5.8 & S. 289, Kap. 8.3.1]

Die Auswahl einer geeigneten „Clock-Source“ für den Timer erfolgt über spezifische Bits im *TAxCTL* oder *TBxCTL* Timer-Control-Register. Das *TASSEL-/TBSSEL*-Bit legt fest, ob der Timer von *TAxCLK/TBxCLK*, *ACLK*, *SMCLK* oder *INCLK* getaktet wird. Diese Wahl hat einen direkten Einfluss auf die Timer-Frequenz, wobei sie nicht gleich der Frequenz der gewählten Clock-Source entsprechen muss. Durch optionale *Prescaler-Werte*<sup>17</sup> wie dem *ID*-Bit und dem *TAIDEX-/TBIDEX*-Bit kann die Frequenz weiter individualisiert werden.

[11, 4, S. 349, Kap. 11.3.1 & S. 372, Kap. 12.3.1, S. 289, Kap. 8.3.1]

Die Timer-Frequenz bestimmt wiederum die Zeitbasis des Timers. Eine höhere Timer-Frequenz führt zu einer feineren Zeitauflösung, da der Zähler schneller inkrementiert. Dies ermöglicht präzise Zeitmessungen und die Erzeugung hochfrequenter Signale. Umgekehrt führt eine niedrigere Frequenz zu einer gröberen Zeitauflösung, kann aber den Stromverbrauch reduzieren.

Ein weiteres Steuerbit, das „Mode Control“-Bit (*MC*) steuert die – bereits in Kapitel 2.1.1 erläuterten – Zähl-Modi und das *TAIE-/TBIE*-Bit steuert, ob Interrupts Ein- oder Ausgeschaltet sind.

---

<sup>15</sup>Niederfrequente Taktquelle in Mikrocontroller-Systemen, die typischerweise von einem Quarzoszillator gespeist wird und für energiesparende Betriebsmodi verwendet wird.

<sup>16</sup>Taktgesteuertes Signal, das typischerweise für Peripheriegeräte verwendet wird und sich aus einer frei wählbaren Taktquelle ableiten lässt.

<sup>17</sup>Vorschaltglied in elektronischen Zählschaltungen oder Timern, welches die Frequenz eines Eingangssignals durch einen festen Faktor reduziert, um eine nachfolgende Verarbeitung mit geringerer Taktrate zu ermöglichen.

Die Auswahl der Clock-Source, des Prescalers und weiteren Steuerbits ist daher entscheidend, um die gewünschte Zeitbasis, Auflösung und Verhalten für den zu konfigurierenden Timer zu erreichen um die Anforderungen der jeweiligen Anwendung optimal zu erfüllen.

Tabelle 2.2 fasst alle weiteren Register des Timers B mit ihren Beschreibungen zusammen.<sup>18</sup>

Bit	Feld	Typ	Reset	Beschreibung
15	Reserved	R	0h	Reserved. Immer als 0 gelesen.
14–13	TBCLGRP	RW	0h	<b>TBxCLn-group</b> ; Synchroner Aktualisierung mehrerer Capture/Compare-Register.
12–11	CNTL	RW	0h	Counter length
10	Reserved	R	0h	Reserved. Immer als 0 gelesen.
9–8	TBSSEL	RW	0h	clock source select
7–6	ID	RW	0h	<b>Input divider</b> ; Teilung der gewählten Clock-Source in Verbindung mit dem TBIDEX-Register.
5–4	MC	RW	0h	<b>Mode control</b> ; Ausführlicher in Kapitel 2.1.1
3	Reserved	R	0h	Reserved. Immer als 0 gelesen.
2	TBCLR	RW	0h	Setzt TBR und Kontroll-Logik zurück.
1	TBIE	RW	0h	Timer_B interrupt enable
0	TBIFG	RW	0h	Timer_B interrupt flag

**Tab. 2.2.:** Registerbeschreibung – Control Register Timer B  
[11, S. 372, Tab. 12-6]

<sup>18</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

### 2.1.6. Zusammenfassung der Einsatzmöglichkeiten

Die detaillierte Auseinandersetzung mit der Timer-Architektur des MSP430FR5729 verdeutlicht die Flexibilität und Leistungsfähigkeit dieser Peripheriekomponente. Die Unterscheidung zwischen Timer des Typs A und B, die verschiedenen Betriebsarten (Zählmodus und Capture/Compare) sowie die vielfältigen Einstellmöglichkeiten der Steuerregister und die Auswahl der Taktquelle eröffnen ein breites Spektrum an Anwendungsmöglichkeiten in eingebetteter Software.

Analog zur Übersicht „What Timer Where?“ von John H. Davies lassen sich die primären Einsatzgebiete der Timer des MSP430FR5729 wie folgt zusammenfassen: [4, S. 356, Kap. 8.10]

- **Zeitmessung und Zeitbasis:** Unabhängig vom Timer-Typ können alle als eine präzise Zeitbasis dienen. Durch die Wahl einer geeigneten Clock-Source und passender Prescaler-Werte lassen sich genaue Zeitintervalle festlegen. Dies ist fundamental für das Zeitmanagement innerhalb des Mikrocontrollers und die Synchronisation mit externen als auch Internen Ereignissen. Timer A eignet sich für grundlegende Zeitsteuerungsaufgaben, während Timer B durch seine flexiblere Konfiguration eine genauere Anpassung ermöglicht.
- **Ereigniserfassung (Capture):** Die Capture-Funktionalität ermöglicht eine präzise Erfassung des Zeitpunkts externer – oder ggf. auch interner – Ereignisse. Diese Erfassung spielt eine zentrale Rolle bei Anwendungen wie der Pulsweitenmessung, der Frequenzmessung oder der Detektion von Ankunftszeiten. Die Möglichkeit, sowohl steigende, fallende Flanken oder auch beide zu erfassen, erweitert den Anwendungsbereich in vielen verschiedenen Szenarien.
- **Signalerzeugung (Compare/PWM):** Die Compare-Einheiten in Verbindung mit den verschiedenen Ausgangsmodi erlauben die Generierung präziser Ausgangssignale. Dies ist besonders relevant für die Pulsweitenmodulation, die zur Steuerung von Motoren, zur Dimmung von LEDs oder zur Erzeugung analog wirkender Signale eingesetzt wird. Der Up/Down Mode des Count-Modus in Kombination mit den Compare-Registern des Timer B bietet hierbei besonders flexible Möglichkeiten zur Erzeugung verschiedenster PWM-Signale.

- **Interrupt-Steuerung und Task-Rotation:** Sowohl Capture- als auch Compare-Ereignisse lösen Interrupts aus. Dies ermöglicht eine effiziente Reaktion des Mikrocontrollers auf zeitgesteuerte Ereignisse oder Signale, ohne die kontinuierliche Abfrage des Timer-Status. Die präzise Interrupt-Generierung trägt maßgeblich zur Realisierung reaktiver und effizienter eingebetteter (Echtzeit-) Systeme bei.

Zusammenfassend zeigt sich der grundlegende Aufbau eines Timers in Anlehnung an Abbildung 8.5 und Abbildung 8.16 aus Davies' Buch, wie folgt:

Ein Timer besteht im Kern aus einem Zähler (Kapitel 2.1.1), der durch eine ausgewählte Clock-Source (Kapitel 2.1.5) in definierten Schritten inkrementiert oder dekrementiert wird. Der Zähler arbeitet entsprechend der gewählten Betriebsart.

Zusätzlich verfügt der Timer über sieben Capture/Compare-Kanäle. Jeder Kanal beinhaltet mindestens ein Capture/Compare-Register sowie die zugehörige Steuereinheit.

Im Capture Mode (Kapitel 2.1.2) wird der aktuelle Wert des Zählers in das CCRx-Register geschrieben, wenn ein durch die Steuereinheit ausgewähltes Ereignis (z. B. Flanke an einem Eingangspin) eintritt.

Im Compare Mode (Kapitel 2.1.3) wird der aktuelle Wert des Zählers kontinuierlich mit dem Wert im CCRx-Register verglichen. Bei einer Übereinstimmung löst die Steuereinheit eine konfigurierte Aktion aus – wie beispielsweise das Setzen/Rücksetzen/Toggeln eines zugehörigen Ausgangspins oder die Generierung eines Interrupts – sofern der Interrupt in der Steuereinheit aktiviert ist.

Die Steuereinheit ermöglicht die Konfiguration des jeweiligen Kanals, einschließlich der Auswahl des Capture/Compare-Modus, des auslösenden Ereignisses, des Ausgangsmodus und der Aktivierung/Deaktivierung des Interrupts.

Das Blockdiagramm in Abbildung 2.8 zeigt Timer B mit seinen grundlegenden Komponenten und deren Zusammenspiel. Die flexiblen Konfigurationsmöglichkeiten dieser Blöcke ermöglicht die Realisierung einer Vielzahl von Zeitsteuerungs- und Signalverarbeitungsaufgaben.

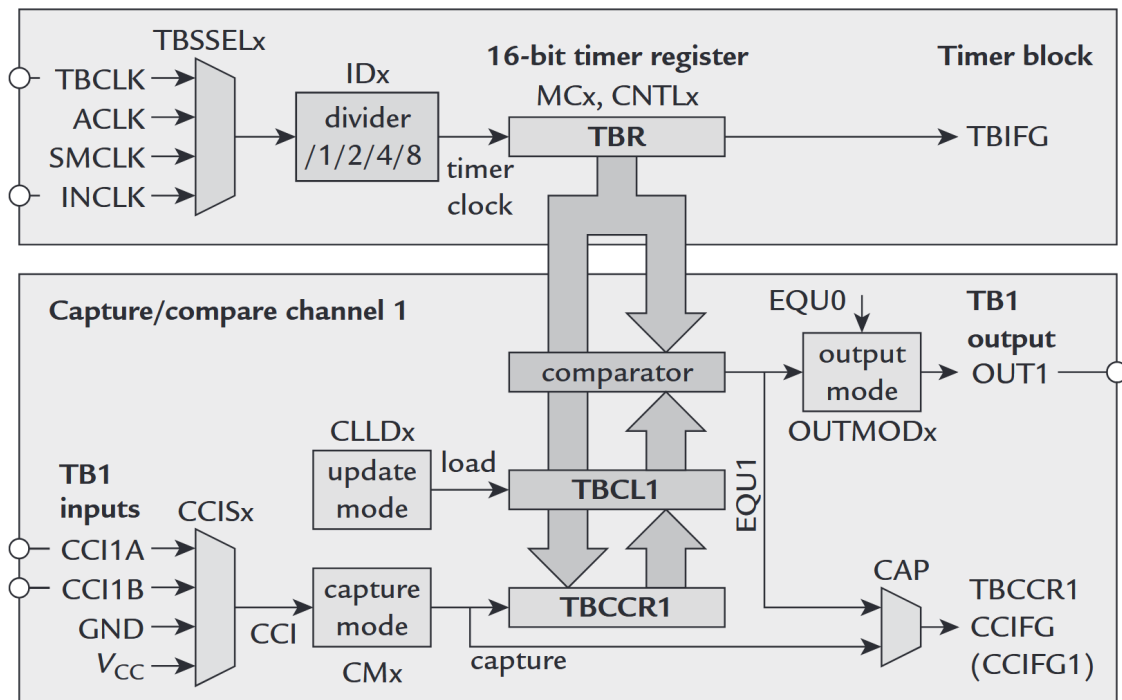


Abb. 2.8.: Timer B Block & Capture/Compare Channel 1  
[4, S. 355, Kap. 8.16]

## 2.2. Enhanced Universal Serial Communication Interface (eUSCI)

Das eUSCI ist eine vielseitige und flexible serielle Peripheriekomponente des MSP430FR5729. Sie ermöglicht die Kommunikation mit externen Geräten und Systemen über zahlreiche Schnittstellen und Protokolle. Auch Sensoren, Aktoren und Speichermedien verbindet das System auf diese Weise. Dieses Kapitel beleuchtet die Architektur des *Interfaces*<sup>19</sup>, seine verschiedenen Betriebsmodi und Konfigurationsmöglichkeiten bereitgestellter Kommunikationstechnologien.

Der MSP430FR5729 besitzt zwei Kommunikationskanäle, die die folgenden Kapitel näher beschreiben.

<sup>19</sup>Ein Interface bezeichnet in der Informatik eine definierte Schnittstelle, über die Komponenten eines Softwaresystems miteinander kommunizieren können, ohne deren interne Implementierung zu kennen.

### 2.2.1. Die eUSCI-Architektur

Jede Form der *Seriellen Kommunikation*<sup>20</sup> basiert auf einem Taktgeber. Der zentrale Unterschied zwischen Protokollen liegt im Timing: Wann gestattet der Taktgeber dem Sender das Schreiben des nächsten Bits auf einen Ausgangskanal und dem Empfänger das Lesen des kommenden Bits. Dabei gibt es Synchrone und Asynchrone Ansätze, weshalb zwei Kommunikationskanäle bereitgestellt werden.

[4, S. 494, Kap. 10]

Beim MSP430FR5729 ist der Kommunikationskanal vom Typ-B für Synchrone Datenübertragung optimiert, während Kanal A vorrangig für asynchrone Übertragungsverfahren vorgesehen ist. [4, S. 496, Kap. 10.1.2]

Der wesentliche Unterschied zwischen diesen beiden Übertragungsarten besteht darin, ob das Taktsignal ebenfalls mit übertragen wird. Bei der synchronen Kommunikation, wie sie etwa mit den Protokollen SPI oder I<sup>2</sup>C erfolgt, wird dieses Taktsignal explizit mitgeführt. Im Gegensatz dazu kommt das UART-Protokoll ohne ein separates Taktsignal aus, da Sender und Empfänger über eine gemeinsame *Baudrate*<sup>21</sup> synchronisiert sind. [4, S. 494, Kap. 10]

Entsprechend ist der universelle Kommunikationskanal vom Typ B (*eUSCI\_B*) speziell auf die Anforderungen synchroner Protokolle wie SPI und I<sup>2</sup>C ausgelegt. Das *eUSCI\_A*-Modul hingegen unterstützt primär die UART-Kommunikation, kann jedoch darüber hinaus auch eine asynchrone Variante des SPI-Protokolls abbilden.

[4, S. 496, Kap. 10.1.2]

---

<sup>20</sup>Die Übertragung von Datenbitfolgen über eine einzelne Leitung, wobei die Bits nacheinander (also „seriell“) gesendet werden. Sie wird häufig bei einfachen oder ressourcenschonenden Datenverbindungen eingesetzt, z. B. über UART, SPI oder I<sup>2</sup>C.

<sup>21</sup>Bezeichnet die Anzahl der Signaländerungen (Symbole) pro Sekunde in einem Übertragungskanal. Sie bestimmt somit die Übertragungsgeschwindigkeit, ist aber nicht zwingend identisch mit der Bitrate, da pro Symbol mehrere Bits codiert sein können. [21]



Funktion	eUSCI_A	eUSCI_B
UART (asynchron)	✓	–
SPI (synchron)	✓ (Master/Slave)	✓ (Master/Slave)
SPI (asynchron, nur TX)	✓	–
I <sup>2</sup> C (synchron)	–	✓ (Master/Slave)
LIN-kompatibel ( <i>Local Interconnected Network</i> <sup>22</sup> )	✓	–
Automatische Baudratenerkennung (UART)	✓	–
Adress- und Broadcast-Modus (I <sup>2</sup> C)	–	✓
Multimaster-Unterstützung (I <sup>2</sup> C)	–	✓

**Tab. 2.3.:** Funktionsvergleich der eUSCI-Module des MSP430FR5729  
[11, 4, Kap. 18, 19, 20, S. 493, Kap. 10]

Tabelle 2.3 fasst nochmals alle Eigenschaften beider Kanäle zusammen und stellt sie gegenüber. Wobei tiefgreifendere protokollspezifische Funktionen in den entsprechenden Unterkapiteln näher betrachtet werden.

### 2.2.2. Einordnung vorhandener Kommunikationsschnittstellen

Diese Arbeit entwickelt eine interruptgesteuerte Benutzerschnittstelle. Die Evaluations dafür geeigneter Protokolle zur Interaktion mit externen *PC* Systemen bestimmt maßgeblich den weiteren Verlauf des Projekts. Daher ordnet die Arbeit die verfügbaren seriellen Protokolle ein, um im weiteren Verlauf gezielt auf jene Technologie eingehen zu können, welche im Kontext der Arbeit von praktischer Relevanz ist.

Die Kommunikation über synchrone Protokolle wie I<sup>2</sup>C und SPI eignen sich besonders gut für den Datenaustausch zwischen einem Mikrocontroller und seinen Peripheriegeräten oder weiteren Mikrocontrollern im Master-Slave-Verhältnis. Die Wahl der Technologie richtet sich unter anderem nach der Anzahl beteiligter Geräte sowie der Distanz zu den Kommunikationspartnern. Weitere technische Unterschiede dieser Protokolle sind in Tabelle 2.4 aufgelistet.

<sup>22</sup>Serieller Kommunikationsstandard, zur kostengünstigen Vernetzung von Mikrocontrollern in Fahrzeugen, insbesondere bei weniger zeitkritischen Steuergeräten. [20]

Kriterium	SPI	I <sup>2</sup> C
Signalleitungen	4 Leitungen: SCLK, MOSI, MISO, CS (pro Slave)	2 Leitungen: SCL (Takt), SDA (Daten)
Adressierung	Keine; Slaves über eigene Chip Selects (CS)	Ja; über 7- oder 10-Bit-Adresse auf dem Bus
Datenübertragung	<i>Vollduplex</i> <sup>23</sup> möglich	<i>Halbduplex</i> <sup>24</sup>
Taktfrequenz	Bis > 10 MHz (geräteabhängig)	Typisch 100 kHz, 400 kHz, bis 3.4 MHz (High-Speed)
Komplexität des Protokolls	Einfach, ohne Start-/Stopp- oder ACK-Signale	Höher, mit Start-/Stoppbedingungen und Acknowledgements
Multimaster-Unterstützung	Nein (standardmäßig)	Ja
Skalierbarkeit (Anzahl Geräte)	Eingeschränkt, abhängig von verfügbaren CS-Leitungen	Hoch, bis zu 128 Geräte durch Adressierung
Typische Einsatzgebiete	Hochgeschwindigkeitskommunikation (z. B. SD-Karten, Displays)	Niedriggeschwindigkeitskomponenten (z. B. Sensoren, EEPROMs)
Leitungslänge / Störanfälligkeit	Gut für kurze, direkte Verbindungen	Höhere Anfälligkeit für Störungen und Begrenzung durch Leitungskapazität

**Tab. 2.4.:** Vergleich der synchronen seriellen Protokolle SPI und I<sup>2</sup>C

[4, S. 497, Kap. 10.2, S. 534, Kap. 10.7]

Zusammenfassend zeigt sich, dass die synchrone Datenübertragung sich nur bedingt für die Kommunikation mit einem auf Windows oder Linux basierenden System eignet. Im Gegensatz dazu sind asynchrone Protokolle wie UART für diese Art der Anwendung deutlich besser geeignet, wie der folgende Abschnitt beschreibt.

Das UART-Protokoll zeichnet sich nicht nur durch eine einfache Implementierung aus, sondern ist auch äußerst robust und ressourcenschonend – Eigenschaften, die insbesondere bei modularen, *Plug-and-Play-fähigen* <sup>25</sup> Systemkomponenten entscheidend sind. Der Verzicht auf eine gemeinsame Taktleitung erlaubt eine Punkt-zu-Punkt-Verbindung mit vergleichsweise geringen Hardwareanforderungen.

<sup>23</sup> Gleichzeitige Datenübertragung in beide Richtungen. [18]

<sup>24</sup> Datenübertragung zu einem Zeitpunkt nur in eine Richtung möglich. [18]

<sup>25</sup> Automatische Erkennung und Integration von Komponenten in ein System ohne manuelle Konfiguration. [19]

Diese Vorteile gelten gleichermaßen für die gegenüberliegende Seite der Schnittstelle: Alle gängigen Betriebssysteme wie Windows, Linux oder macOS stellen standardmäßig Treiber für die asynchrone serielle Datenübertragung bereit. Unter Windows erfolgt dies beispielsweise über sogenannte **COM-Ports**, während unter Linux Schnittstellen wie `/dev/ttySx` oder `/dev/ttyUSBx` verwendet werden. [16, 2]

Im Falle des MSP430FR5729 erfolgt die UART-Kommunikation zusätzlich interruptgesteuert. Dies erlaubt es, eingehende Daten über eine eigens definierte ISR zu verarbeiten, was eine latenzarme, gleichzeitig jedoch energieeffiziente Verarbeitung ermöglicht. [4, S. 574, Kap. 10.12]

Aus diesen Gründen stellt UART die technisch sinnvollste Wahl für die Kommunikation zwischen dem MSP430FR5729 und einem PC mit Windows, Linux oder macOS dar. Das Protokoll erlaubt eine minimalinvasive, betriebssystemkompatible und energieeffiziente Verbindung. Im weiteren Verlauf wird die asynchrone universelle serielle Schnittstelle mit dem UART Protokoll detaillierter betrachtet.<sup>26</sup>

### 2.2.3. eUSCI\_A Modul: UART-Modus (Asynchrone Kommunikation)

Für ein tiefgreifendes Verständnis des Zusammenspiels zwischen dem Interface und dem UART-Protokoll ist es unerlässlich, zunächst die fundamentalen technischen Aspekte, notwendige Register und charakteristische Merkmale zu erläutern.

#### 2.2.3.1. Informationsübertragung

Die Baudrate, wie in Kapitel 2.2.1 bereits erörtert, fungiert als entscheidender Synchronisationsmechanismus für asynchrone Datenübertragungen. Dies impliziert, dass Sender und Empfänger sich zwar nicht an ein präzises Timing für die Übertragung einzelner Bits halten müssen, jedoch eine Übereinstimmung hinsichtlich der Frequenz zur Übertragung ganzer Blöcke – typischerweise Bytes oder Zeichen – erforderlich ist.

---

<sup>26</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

Abbildung 2.9 visualisiert beispielhaft die Übertragung zweier Blöcke, die jeweils durch ein Start-Bit (*ST*) eingeleitet und durch ein Stop-Bit (*SP*) abgeschlossen werden. Durch die Verwendung dieser Rahmenbits ergibt sich bei einer Konfiguration von acht Datenbits eine Netto-Datenrate von 8/10 der Brutto-Übertragungsrate. Das bedeutet, dass von zehn übertragenen Bits acht Bits die eigentliche Nutzinformation darstellt. Es gibt auch die Möglichkeit, flexibel auf unterschiedliche Baudraten zu reagieren. Vorausschauend sei erwähnt, dass die in Kapitel 2.2.3.3 erläuterte Automatische Baudratenerkennung die Kommunikation mit mehreren Partnern bei unterschiedlichen Baudraten erlaubt. [4, S. 576, Kap. 10.12.1]

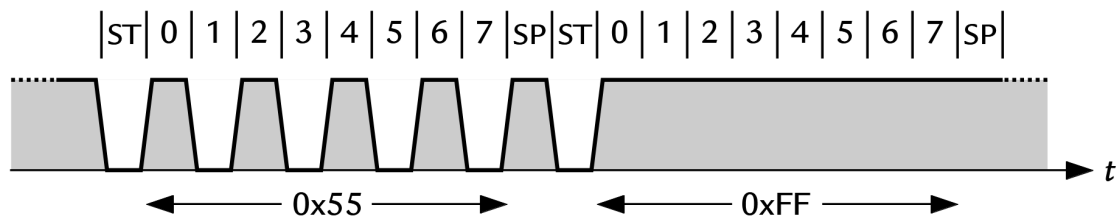


Abb. 2.9.: UART übertragung der Werte 0x55 und 0xFF  
[4, S. 576, Abb. 10.18]

Die einzelnen Bits innerhalb eines Datenblocks werden mittels des *Non-Return-to-Zero-Verfahrens*<sup>27</sup> (*NRZ*) kodiert und übertragen. Eine typische Baudrate für eingebettete Systeme beträgt 9600 Baud, obgleich auch höhere Frequenzen zur beschleunigten Datenübertragung Anwendung finden können. [21]

Drei Leitungen stellen üblicherweise die physikalische Verbindung zwischen zwei Parteien her. Eine Leitung für jede Kommunikationsrichtung (*TxD* zu *RxD*) und eine für die gemeinsame Masse. Dies ermöglicht eine **Vollduplex-Kommunikation**, bei der beide Seiten gleichzeitig und unabhängig voneinander Daten senden und empfangen können. Voraussetzungen hierfür sind separate Sende-/Empfangsschieberegister sowie dedizierte Puffer – *UCAxRXBUF* als Empfangs- und *UCAxTXBUF* als Sendepuffer – für beide Kommunikationsrichtungen in der Hardware des Interfaces.

[11, 6, S. 499, Kap. 18.4.6 & 18.4.7]

<sup>27</sup>Binäres Leitungscodierungsverfahren, bei dem der Signalpegel während eines Bitintervalls konstant bleibt und nicht zwischen den Bits auf einen Nullpegel zurückkehrt. [9]

Beim Empfang eines UART-Blocks ergibt sich folgender Ablauf:

1. Beginn der Zeitmessung mit der fallender Flanke, die das Startbit einleitet.
2. Abtastung des Eingangs nach einer halben Bitperiode zur Bestätigung eines gültigen Startbits.
3. Weitere Abtastung nach einer vollständigen Bitperiode zur Erfassung des ersten Datenbits (*LSB*).
4. Wiederholung dieses Vorgangs für alle 8 Datenbits bis zum höchstwertigen Bit (*MSB*).
5. Abschließende Abtastung nach einer weiteren Bitperiode zur Überprüfung des Stopbits (*High-Pegel*<sup>28</sup> erwartet). Liegt stattdessen ein *Low-Pegel*<sup>29</sup> vor, wird ein Framing-Fehler erkannt.

Abbildung 2.10 visualisiert diesen Empfangsprozess unter Verwendung einer sogenannten „*sampling clock*“<sup>30</sup>. Diese Abtastfrequenz ist üblicherweise um den Faktor 16 höher als die konfigurierte Baudrate. Das *Oversampling*<sup>31</sup> ist notwendig um das eintreffende Start-Bit zuverlässig und zeitnah auch zwischen den regulären Bit-Takten detektieren zu können.

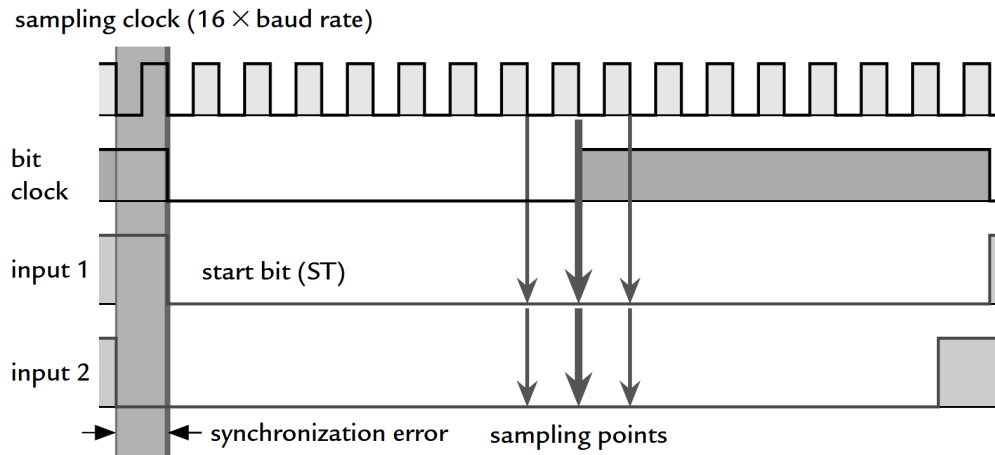
---

<sup>28</sup>Bezeichnet in der Digitaltechnik einen Spannungszustand, der über einer definierten Schaltschwelle liegt. [7]

<sup>29</sup>Bezeichnet in der Digitaltechnik einen Spannungszustand, der unter einer definierten Schaltschwelle liegt. [7]

<sup>30</sup>Bestimmt den Zeitpunkt der Abtastung eingehender Bits; sie wird üblicherweise aus einer übergeordneten Taktquelle (z. B. SMCLK) abgeleitet und beeinflusst maßgeblich die Genauigkeit der Datenübertragung. [4]

<sup>31</sup>Oversampling bezeichnet das Verfahren, bei dem ein Signal mit einer höheren Abtastrate als der Abtastgrenze (Nyquist-Rate) digitalisiert wird, um Rauscheinflüsse zu reduzieren und die Signalqualität zu verbessern. [4]



**Abb. 2.10.:** UART Übertragung der Werte 0x55 und 0xFF  
[4, S. 577, Abb. 10.19]

Die interne Bit-Clock (*BITCLK*) des Empfängers wird mit der fallenden Flanke des eingegangenen Start-Bits synchronisiert und operiert mit der Frequenz der eingestellten Baudrate. Da die fallende Flanke des Start-Bits zu einem beliebigen Zeitpunkt relativ zur Sampling Clock auftreten kann, entsteht ein initialer Synchronisationsfehler von bis zu einer halben Periode der Sampling Clock. Die in Abbildung 2.10 dargestellten Szenarien, bezeichnet als „Input 1“ und „Input 2“, illustrieren die hieraus resultierende minimale und maximale zeitliche Verschiebung bei der Detektion der Startbit-Flanke, abhängig vom Phasenverhältnis zwischen dem Datensignal und der Sampling Clock.

[11, 4, S. 476, kap. 18.2, S. 574, Kap. 10.12 & S. 575, Kap. 10.12.1]

### 2.2.3.2. Datenintegrität, Fehlererkennung und weitere technischen Details

Zur Sicherstellung der Datenintegrität kann eine Fehlererkennung, beispielsweise über ein Paritätsbit, eingesetzt werden. Ein UART-Datenpaket, auch Frame genannt, besteht typischerweise aus einem Start-Bit, sieben oder acht Datenbits, einem optionalen Paritätsbit (für gerade oder ungerade Parität) und mindestens einem Stop-Bit.

Die übergeordnete Protokollebene implementiert üblicherweise komplexere Fehlererkennung oder -korrekturmechanismen, wie z. B. Prüfsummen (Checksum), die auf der UART-Kommunikation aufsetzt. Darüber hinaus ist für eine erfolgreiche Kommunikation die eindeutige Festlegung der Bitreihenfolge essentiell. Die LSB-first-Konvention ist De-facto-Standard. [4, S. 574, Kap. 10.12 & S. 575, Kap. 10.12.1]

Die automatische Fehlererkennung des Interfaces ermöglicht dem Benutzer eine schnelle Reaktion auf Grenzfälle und Übertragungsfehler. Tabelle 2.5 schlüsselt alle wichtigen Fehler-Flags mit ihren zugehörigen Beschreibungen auf.

Fehlerbedingung	Fehler-Flag	Beschreibung
Framing-Fehler	UCFE	Tritt auf, wenn das Stoppbit nicht den erwarteten High-Pegel hat. Bei zwei Stoppbits werden beide geprüft.
Paritätsfehler	UCPE	Entsteht durch eine Abweichung zwischen berechneter und tatsächlicher Parität. Adressbits werden in die Berechnung einbezogen.
Empfangsüberlauf	UCOE	Wenn ein neues Zeichen empfangen wird, bevor das vorherige gelesen wurde, wird ein Überlauf erkannt.
Break-Bedingung	UCBRK	Wird erkannt, wenn alle Bits auf Low liegen (bei deaktivierter Baudratenerkennung). UCBRK wird gesetzt und ggf. auch UCRXIFG, wenn UCBRKIE aktiv ist.

**Tab. 2.5.:** UART-Fehlerbedingungen und zugehörige Status-Flags des MSP430FR5729 [11, S. 483, Tab. 18-1]

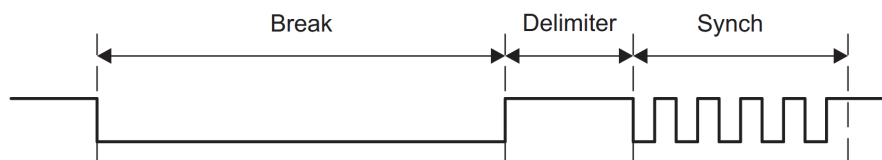
Weitere technische Spezifikationen sind in Tabelle 2.6 zusammengefasst. Eine minimale Konfiguration der Schnittstelle auf den UART-Betrieb wird in Kapitel 2.2.4 detailliert beschrieben.

Funktion	Beschreibung
Multiprozessor-Kommunikationsprotokolle	Unterstützt integrierte Idle-Line- und Address-Bit-Protokolle für Kommunikation in Multiprozessorsystemen
Energiesparmodus-Unterstützung	Startflankenerkennung im Empfänger ermöglicht automatisches Aufwachen aus LPMx-Modi (ausgenommen LPMx.5)
Fehlererkennung	Statusflags zur Detektion von Kommunikationsfehlern (z. B. Framing-, Paritäts- oder überlauffehler)
Adresserkennung	Statusflags zur Erkennung von Datenpaketen in Multiprozessorsystemen
Interrupt-Unterstützung	Unabhängige Interruptquellen für Empfang, Übertragung, Startbit-Empfang sowie Abschluss der Übertragung

**Tab. 2.6.:** Technische Merkmale der UART-Schnittstelle des MSP430FR5729 [11, S. 476, kap. 18.2]

### 2.2.3.3. Automatische Baudratenerkennung

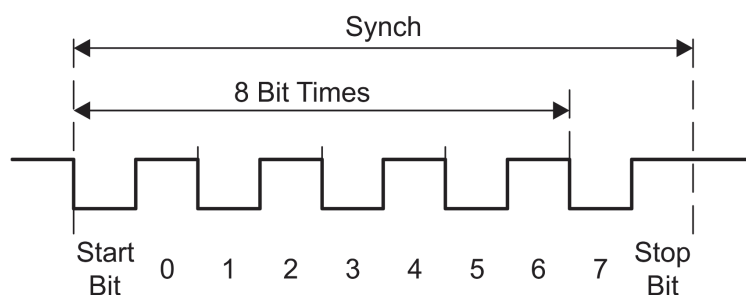
Neben der festen Einstellung ermittelt die Automatische Baudratenerkennung selbstständig, über eine **Break/Sync Sequenz**, die vom Sender verwendete Baudrate. Diese Synchronisations-Sequenz besteht aus einem **Break** und einem **Sync** Feld. Der Bereich der erkennbaren Baudraten liegt im Oversampling-Modus zwischen 244 Baud (im niedrigfrequenz-Modus beginnend ab 15 Baud) und einem Megabaud. Ein Break umfasst 11 bis 21 übertragen Nullen, während weitere Nullen einen **Break Timeout**-Fehler auslösen. Aus Konformitätsgründen sollte das UART Protokoll auf acht Datenbits, mit LSB-first, keiner Parität und einem Stop-Bit konfiguriert werden. In Abbildung 2.11 ist die beschriebene Break/Sync-Sequenz dargestellt.



**Abb. 2.11.:** Automatische Baudratenerkennung - Break/Sync Sequenz  
[11, S. 481, Abb. 18-5]

Der Synchronisations-Prozess beginnt mit der Übertragung des hexadezimalen Wertes 0x55. Die Zeit zwischen der ersten und letzten fallenden Flanke wird gemessen, um die vom Sender verwendete Baudrate zu ermitteln. Dies ist grafisch in Abbildung 2.12 dargestellt. Überschreitet die Messzeit den Maximalwert, tritt ein **Sync-Timeout**-Fehler auf. Ist die Messung erfolgreich, liest das System nach dem Setzen des **Receive Interrupt Flags** die Information aus.

Nach jedem empfangenen Zeichen setzt das System das **UCDORM**-Bit zurück, da bei gesetztem Bit zwar alle Zeichen empfangen, aber nicht in das Puffer-Register der Schnittstelle geschrieben werden. [11, S. 481, Kap. 18.3.4]



**Abb. 2.12.:** Automatische Baudratenerkennung - Sync Feld  
[11, S. 481, Abb. 18-6]



### 2.2.4. eUSCI-Konfiguration

Die Initialisierung und Konfiguration des eUSCI\_A-Moduls für den UART-Betrieb folgt einer definierten Abfolge von Schritten zum Setzen von verschiedenen Bits und Werten in die dafür vorgesehene Register. Der Prozess setzt das UCSWRST-Bit zu Beginn. Dieses Bit ermöglicht die Konfiguration der Schnittstelle und schließt unerwünschtes Verhalten aus. Dabei wird das UCTXIFG-Bit, zum freigeben der Konfiguration, gesetzt. Zudem werden diverse Interrupt-Enable-Bits wie UCRXIE und UCTXIE sowie Status- und Fehlerflags (UCRXIFG, UCRXERR, UCBRK, UCPE, UCOE, UCFE, UCSTOE, UCBTOE) im UCAxSTATW- und UCAxIFG-Register gelöscht oder in einen definierten Anfangszustand gebracht. Dadurch gelangt das eUSCI\_A-Modul in einen sicheren Reset-Zustand. [11, S. 478, Kap. 18.3.1]

Nachdem das Modul sicher in den Reset-Zustand überführt wird, lassen sich weitere spezifische Konfigurationsparameter für den UART-Betrieb setzen. Hierzu zählen insbesondere die folgenden Kontrollbits im UCAxCTLWn-Register, welche die grundlegenden Betriebscharakteristika des UART-Modus definieren, wie beispielsweise:

- **UCPEN:** Aktiviert die Paritätsprüfung.
- **UCPAR:** Festlegen einer geraden oder ungeraden Parität.
- **UCMSB:** Legt die Bitreihenfolge fest (LSB- oder MSB-first).
- **UC7BIT:** Konfiguriert die Datenlänge auf 7 oder 8 Bit.
- **UCSPB:** Anzahl der Stop-Bits.
- **UCMODEx:** Wählt den UART-Modus. (z. B. 0 für Normal-Betrieb, 3 für automatische Baudratenerkennung)
- **UCSYNC:** Für den asynchronen UART-Betrieb muss dieses Bit auf 0 gesetzt werden.
- **UCSSELx:** Taktquelle für Baudratengenerator.

Über die genannten grundlegenden Einstellungen hinaus existieren weitere spezifischere Kontrollbits. Beispielsweise ist das UCRXEIE-Bit zur freigabe des Fehler-Interrupts und das UCBRKIE zum aktivieren der Break-Interrupts zuständig.

Spezialfunktionen wie der Multiprozessor-Modus, welcher über das `UCTXADDR`-Bit gesteuert wird, oder das Senden eines Break-Zeichens mittels `UCTXBRK` sind für eine Standard-UART-Konfiguration oft zu vernachlässigen, sofern diese Funktionalitäten nicht explizit gefordert sind. [11, S. 495, Kap. 18.4.1 & S. 496, Kap. 18.4.2]

Eine weitere essenzielle Konfiguration für den UART-Betrieb ist die der Baudrate. Diese erfolgt über das `UCAxBRW`-Register und das `UCAxMCTLW`-Register. Die korrekte Wertermittlung für diese Register ist direkt von der Frequenz der zuvor mittels `UCSSELx` gewählten Taktquelle sowie der angestrebten Baudrate abhängig. Der „Family User’s Guide“ des MSP430FR5729 von Texas Instruments liefert für die Berechnung dieser Werte detaillierte Formeln und Beispieltabellen. Das `UCAxBRW`-Register nimmt den ganzzahligen Anteil des Baudratenteilers (Prescaler) auf. Das `UCAxMCTLW`-Register beinhaltet die Konfiguration für die Modulation der Frequenz, sowie das Bit zur Aktivierung des Oversampling-Modus. Durch das `UCBRFx`-Bit wird, in der ersten Modulationsstufe, die Feineinstellung des Prescalers vorgenommen. In der zweiten Modulationsstufe wird durch `UCBRs` ein Modulationsmuster für die `BITCLK` festgelegt. Im letzten Bitfeld kann nun der Oversampling-Modus aktiviert oder deaktiviert werden. [11, S. 487, Kap. 18.3.10 & S. 497, Kap. 18.4.3, 18.4.4]

Obwohl weitere spezifische Einstellungen möglich sind, würde deren detaillierte Erörterung den Rahmen dieser Übersicht überschreiten. Eine unerlässliche, abschließende Konfigurationsmaßnahme vor der Inbetriebnahme betrifft jedoch die Port-Pins: Die für den UART Betrieb verwendeten Pins müssen, über die Function-Select-Register (`PxSEL`), auf die asynchrone UART-Kommunikation konfiguriert werden.

[11, S. 294, Kap. 8.2.5]

Nach Abschluss aller Konfigurationseinstellungen wird das `UCSWRST`-Bit im `UCAxCTLW0`-Register gelöscht (auf ‘0’ zurückgesetzt). Dieser Schritt hebt den Reset-Zustand auf und aktiviert das eUSCI-Modul mit der zuvor definierten Konfiguration. Optional können nun die gewünschten Interrupts, wie z. B. der Sende- (`UCTXIE`), Empfangs- (`UCRXIE`), Transmit-Complete- (`UCTXCPTIE`) oder Start-Bit-Interrupt (`UCSTTIE`), im `UCAxIE`-Register aktiviert werden, um eine ereignisgesteuerte Datenverarbeitung zu ermöglichen. [11, S. 502, Kap. 18.4.10]

Diese sorgfältige Konfigurationssequenz ist entscheidend für die zuverlässige Funktion der UART-Schnittstelle des MSP430-Mikrocontrollers.

### 2.2.5. Zusammenfassung

Die vorangegangenen Kapitel haben die Kommunikationsschnittstelle des MSP430FR5729 im UART-Betrieb detailliert beleuchtet. Das eUSCI-Modul dient der seriellen Kommunikation mit Peripheriegeräten oder anderen Systemen. Dazu werden synchrone Protokolle wie SPI und I<sup>2</sup>C und asynchrone Protokolle wie UART bereitgestellt.

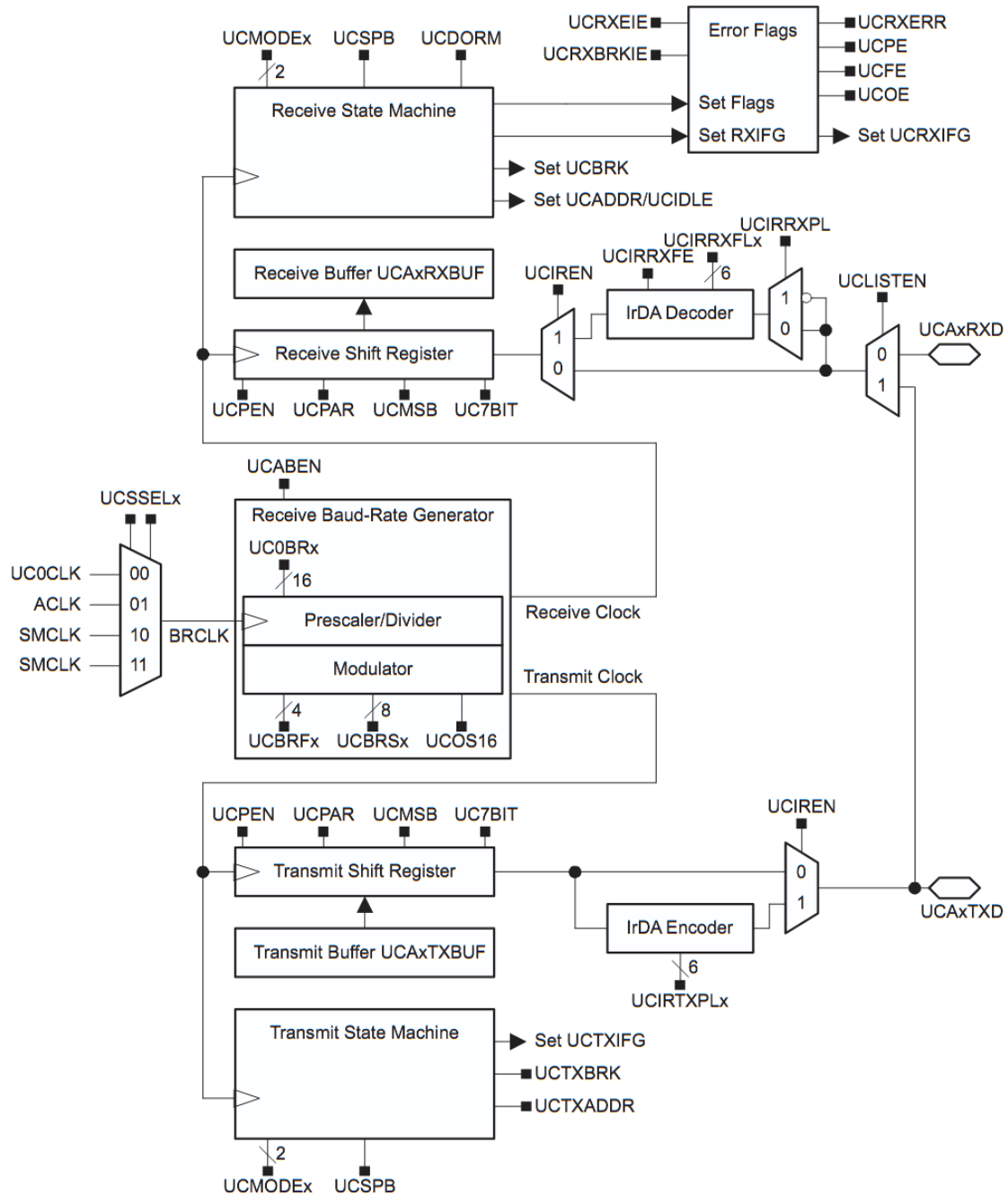
Im Rahmen dieser Arbeit zeigte sich (siehe Kapitel 2.2.2), dass das asynchrone Kommunikationsprotokoll UART am besten für die Kommunikation zwischen dem MSP430FR5729 und einem PC-System geeignet ist. Die Bewertung der Schnittstellen erfolgte unter den Gesichtspunkten Implementierungsaufwand, Robustheit, Interruptfähigkeit und Kompatibilität.

Wesentliche Aspekte des UART-Betriebs ist die asynchrone Natur der Kommunikation und die wechselseitige Datenübertragung zwischen zwei Parteien mittels Baudraten-Synchronisation, auch Vollduplex-Kommunikation genannt. Wichtige Merkmale hierzu ist das Datenformat (z. B. acht Datenbits, LSB-first, ein SP) und die Mechanismen zur Fehlererkennung (z. B. Framing-, Parity-, Overrun-Error).

Die Automatische Baudratenerkennung erhöht bei der Verknüpfung mehrerer Kommunikationspartner über dieselbe Leitung die Flexibilität, steigert jedoch auch die Komplexität. Da für die aktuelle Anwendung keine Verbindung mit unterschiedlichen Baudraten über eine einzelne Schnittstelle erforderlich ist, wird diese Funktion nicht benötigt.

Die zentralen Schritte zur Konfiguration des eUSCI-Moduls für den UART-Betrieb umfasst das Einleiten des eUSCI-Software-Resets, das Setzen und gegebenenfalls Rücksetzen notwendiger Steuerbits in den Kontrollregistern, die Feinabstimmung der gewünschten Baudrate über Prescaler-Werte und schließlich das Aufheben des Reset-Zustandes zur Aktivierung des Moduls.

Abbildung 2.13 zeigt ein detailliertes Blockdiagramm des eUSCI\_A-Moduls in der konfigurierten UART-Betriebsart.



**Abb. 2.13.:** eUSCI Typ A – UART-Modus  
[11, S. 477, Kap. 18.2]

### 3. Planung und praktische Umsetzung

Aufbauend auf den in den vorangegangenen Kapiteln – Kapitel 2.1 und Kapitel 2.2 – vermittelten theoretischen Grundlagen zu Timern, Interrupt Service Routinen und dem Enhanced Universal Serial Communication Interface des Mikrocontrollers MSP430FR5729, widmet sich dieses Kapitel der Entwicklung und praktischen Umsetzung der interruptgesteuerten Benutzerschnittstelle.

Zunächst erfolgt die systematische Konzeptionierung des „Observer-Moduls“. Dieses dient als zentrale Instanz zur Erfassung und Bearbeitung von Befehlseingaben sowie zur Steuerung zugehöriger Systemausgaben. Somit stellt es die Grundlage für die Interaktion zwischen Benutzer und Mikrocontroller dar.

Im weiteren Verlauf wird die Konfiguration der interruptgesteuerten Kommunikation über die UART-Schnittstelle erarbeitet. Hierbei liegt der Fokus auf Zuverlässigkeit und Robustheit sowie deren sicherer Ausgabe unter Echtzeitbedingungen.

Abschließend werden zentrale Debugging-Mechanismen vorgestellt, evaluiert und exemplarisch in Form von softwarebasierten *Breakpoints*<sup>32</sup> implementiert. Ziel ist es, die Robustheit, Echtzeitfähigkeit sowie den funktionalen Umfang der zu entwickelnden Komponente zu validieren und zu evaluieren.

Die Integration von Breakpoints ermöglicht es, gezielt in den Programmablauf eingreifen zu können, ohne den Systemzustand zu beeinträchtigen. Insbesondere zu integrierende Funktionen wie z. B. für das Auslesen und beschreiben von Speicherzellen würden hiervon profitieren. Wird ein Breakpoint erreicht, können diese Befehle kontrolliert und störungsfrei ausgeführt werden, da das Hauptprogramm währenddessen in einen definierten Haltemodus versetzt wird. Auf diese Weise lässt sich eine konsistente Speicherzugriffslogik realisieren, die auch während kritischer Zustände wie Kontextwechsel zuverlässig funktioniert.<sup>33</sup>

---

<sup>32</sup>Bezeichnet in der Softwareentwicklung eine vom Entwickler bewusst gesetzte Unterbrechung im Programmablauf, die typischerweise zur Laufzeit-Debugging-Zwecken verwendet wird. Beim Erreichen dieses Punkts wird die Ausführung des Programms angehalten, sodass der aktuelle Zustand (z. B. Variablen, Stack, Speicher) analysiert werden kann.

<sup>33</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

### 3.1. Konzeptionierung Observer-Modul

Die zentrale Einheit zur Verarbeitung eingehender Befehle und zur Steuerung sämtlicher Funktionen stellt das Observer-Modul dar. Es vereint die Interrupt-Basierte Abarbeitung sowie das Senden von Daten und Empfang externer Steuerbefehle. Die im Kapitel 2.1 und Kapitel 2.2 erläuterten Technologien des MSP430FR5729 bilden hierzu die technische Grundlage.

Alle gewünschten Komponenten und ihre funktionale Zuordnung ist in Abbildung 3.1 zusammengefasst.

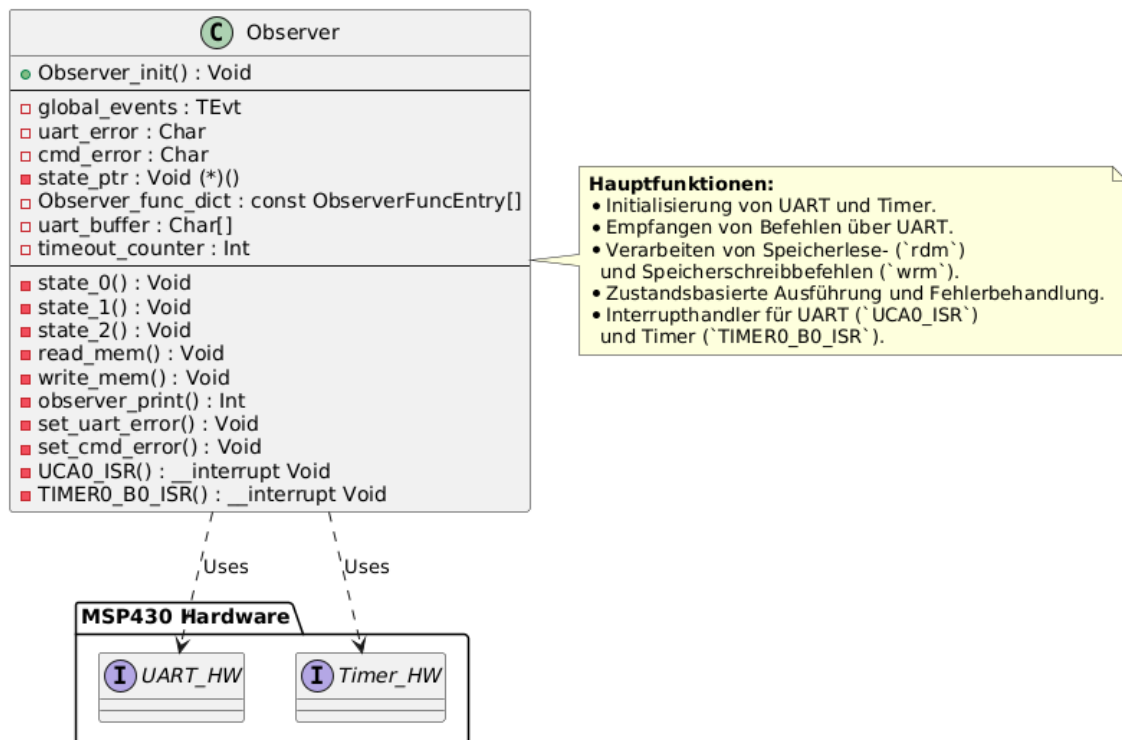


Abb. 3.1.: UML-Diagram – Observer-Modul

Ausgehend vom grundlegenden Architekturkonzept des Observer-Moduls erfolgt in einem kommenden Kapitel – Kapitel 3.3 – eine detaillierte Erklärung der Implementierung der Funktionen für das interruptgesteuerte Lesen und Schreiben über die serielle Schnittstelle.

### 3.1.1. Zustandsautomat und Funktionsumfang

Der Programmablauf, nach der Initialisierung des Moduls bis zur erfolgreichen Ausgabe einer Information, beschreibt sich wie folgt: Solange kein Befehl über die Eingabetaste eines verbundenen PCs bestätigt wurde, verbleibt das Modul im Ruhezustand. Bereits ab dem ersten *alphanumerischen*<sup>34</sup> Zeichen wird ein „Time-Out-Timer“ aktiviert. Dieser Time-Out-Timer bildet den ersten von drei Zuständen (`state_0`). Erfolgt innerhalb eines definierten Zeitintervalls keine abschließende Eingabe, wird der Vorgang über den Zustand `state_2` zurückgesetzt, und es wird ein „Time-Out-Error“ generiert. Bei erfolgreicher Bestätigung hingegen wird der Timer deaktiviert und über den Zustand `state_1` die Befehlsinterpretation eingeleitet. In `state_1` wird ein zum Befehl passender Funktionszeiger dem Statuszeiger zugewiesen. Das Ergebnis der Befehlsverarbeitung wird anschließend über UART zurückgegeben. Etwaige Fehler im Ablauf werden mithilfe eines Fehlervektors priorisiert, gespeichert und entsprechend ausgegeben. Das Observer-Modul arbeitet statusorientiert und verarbeitet Ereignisse gemäß ihres Auftretens. Abbildung 3.2 zeigt einen voll Umfassenden *Zustandsautomaten*<sup>35</sup>. Eine detaillierte Beschreibung der Abarbeitung gesetzter Zustände findet sich in Kapitel 3.1.3.

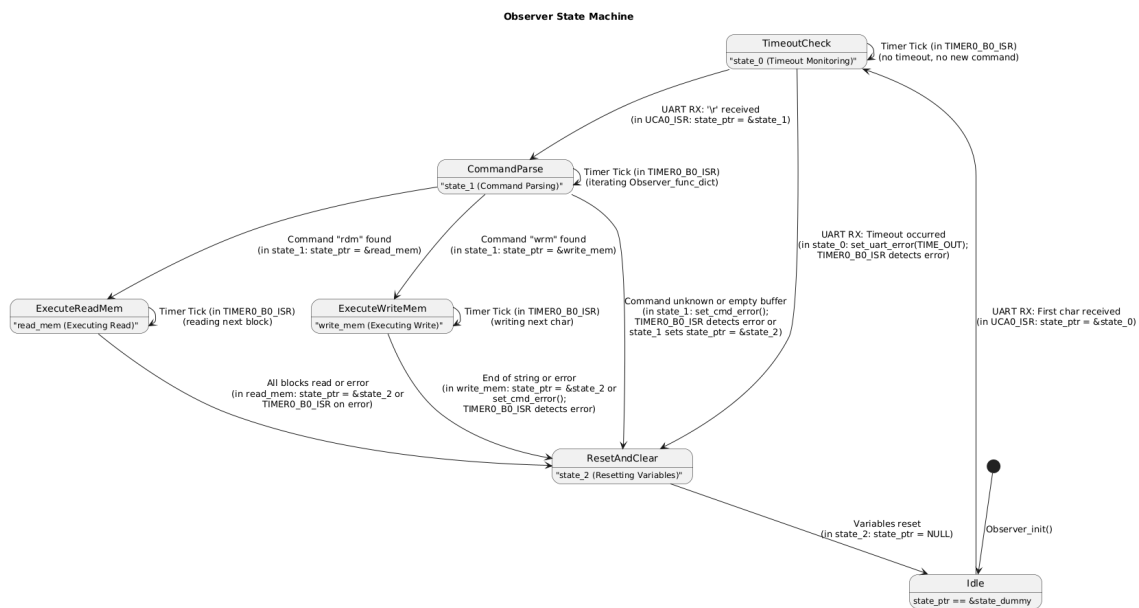


Abb. 3.2.: Zustandsautomat – Observer-Modul

<sup>34</sup>Zeichenkombination aus Buchstaben (lateinisches Alphabet) und Ziffern (0–9).

<sup>35</sup>Auch endlicher Automat, ist ein mathematisches Modell zur Beschreibung von Systemen mit endlich vielen Zuständen, das Übergänge zwischen diesen in Abhängigkeit von Eingaben definiert.

### 3.1.2. Initialisierung

Ein funktionales Modul, welches die dafür benötigten Hardware-Komponenten – insbesondere Interrupt Service Routinen und das eUSCI-Interface – integriert, erfordert eine öffentlich zugängliche Initialisierungsmethode, die von der Hauptroutine aufgerufen werden kann. Diese Methode übernimmt die Konfiguration der Steuerregister sowie die Initialisierung benötigter Variablen und Datenstrukturen.

Für die Ein- und Ausgabe über UART kann die in Kapitel 2.2.4 beschriebene Konfiguration übernommen werden. Folgende Einstellungen haben sich dabei als vorteilhaft erwiesen:

- Gerade Parität
- LSB-first
- Acht-Datenbits
- Ein Stoppbit
- UART-Modus
- Asynchroner Modus
- ACLK-Taktquelle
- Aktiver Fehler-Interrupt
- Aktiver Break-Interrupt
- 9600 Baud
- 16-faches Oversampling

Die Verarbeitung von Ereignissen wie Befehlsinterpretation, Fehlerausgabe und Eingabe-Timeouts wird über eine gemeinsame Interrupt Service Routine realisiert. Eine Periodendauer von 10 ms erscheint als optimaler Kompromiss zwischen guter Reaktivität und einem nicht-blockierenden Verhalten. Dieser Wert soll zudem sicherstellen, dass genügend Rechenzeit für andere Prozesse übrig bleibt. Eine abschließende Beurteilung der gewählten Zeitspanne findet in Kapitel 4.1.1 statt.

Kapitel 2.1 beschreibt die verfügbaren Timer des MSP430FR5729. Aufgrund der in diesem Zusammenhang dargestellten Eigenschaften erweist sich ein Timer vom Typ B als besonders geeignet, da seine erweiterte Konfiguration – gegenüber des Timers von Typ A – eine skalierbare und anpassungsfähige Implementierung des Moduls im Hinblick auf zukünftige Systemanforderungen erlaubt. Neben den funktionalen Vorteilen ergibt sich die Wahl des Timers auch deshalb, weil im System bereits sämtliche Timer vom Typ A anderweitig genutzt werden.



Die relevante Konfiguration der Capture-/Compare-Register basiert auf folgenden Parametern:

- Compare-Modus
- Input Teiler jeweils auf acht
- UP-Modus
- Compare-Register auf 96 setzen
- ACKL als Taktquelle
- Aktivierter Timer-Interrupt

Der Wert für das Compare-Register ergibt sich aus folgender Formel:

$$\text{Compare-Wert} = \frac{f_{\text{ACLK}} \cdot T_{\text{Periode}}}{\text{Teiler}_1 \cdot \text{Teiler}_2} = \frac{614,4 \text{ kHz} \cdot 10 \text{ ms}}{8 \cdot 8} = 96$$

Weitere Initialisierungen umfassen:

- Puffervariablen zur Zeichenspeicherung,
- Event- und Fehlerflags,
- Laufvariablen und Zeiger zur schnellen Navigation innerhalb von Eingabepuffern,
- sowie Funktionszeiger für die Ereignisverarbeitung.

Anstatt Befehle umständlich über eine **switch-case**-Struktur zu verarbeiten, kommt ein *Dictionary*<sup>36</sup> zum Einsatz. Der Vorteil dabei ist, dass jedem Befehl (Schlüssel) direkt ein Funktionszeiger (Wert) zugeordnet wird. Die Befehlsinterpretation erfordert somit nur noch einen schnellen, iterativen Nachschlagevorgang, um den passenden Funktionszeiger zu ermitteln und im Statuszeiger (**state\_ptr**) für die weitere Abarbeitung in der Timer-ISR zu hinterlegen.

---

<sup>36</sup>Datenstruktur, die Schlüssel-Wert-Paare speichert und schnellen Zugriff auf Werte über ihre eindeutigen Schlüssel erlaubt.

### 3.1.3. Timer ISR

Dieses Unterkapitel thematisiert die Timer-ISR innerhalb des Observer-Moduls. Funktionen im Hinblick auf die Fehlerbehandlung und die Steuerung der Zustandslogik sind darüber Implementiert.

Die Timer-ISR wird periodisch ausgeführt und fungiert als fundamentaler Taktgeber für die operativen Sequenzen des Moduls. Die Frequenz dieses Taktes bestimmt die Reaktionsgeschwindigkeit des Systems auf sowohl externe Ereignisse als auch interne Prozesse. Darunter fällt der Abschluss einer Befehlseingabe oder die Evaluierung und Signalisierung von Fehlerzuständen. Um die Leistung und Reaktivität nicht negativ zu beeinträchtigen, ist der Programmcode innerhalb dieser ISR bewusst minimal und auf Ausführungszeit optimiert. Diese Maßnahme ist kritisch, da Interrupts naturgemäß eine blockierende Wirkung auf den Prozessfluss haben.

Der Ablauf der Timer-ISR ist im Aktivitätsdiagramm in Abbildung 3.3 detailliert dargestellt.

Die ISR wird durch das Setzen des TBIFG-Flags im TB0CTL-Register des Timer B0 ausgelöst, nachdem dieser den in TB0CCR0 definierten Zählwert erreicht hat. Ausführlicher in Kapitel 2.1.4 und Kapitel 2.1.5 besprochen. Die primären Aufgaben der Timer-ISR umfassen:

- **Fehlerbehandlung und -signalisierung:** Bei Eintritt prüft die ISR den globalen Ereignis-Parameter – `global_events` – auf eine registrierte UART- (`UART_ERR`) oder Befehls-Fehler-Flag (`CMD_ERR`). Wird ein Fehler detektiert, initiiert die Routine die Ausgabe der dazugehörigen Fehlermeldung über die UART-Schnittstelle mittels der Funktion `observer_print`, welche in Listing 3.7 aus Kapitel 3.1.4 dargestellt ist. Anschließend wird die entsprechende Fehler-Flag zurückgesetzt und der Zustands-Funktionszeiger `state_ptr` auf die Reset-Funktion `state_2` gesetzt, um das Modul in einen definierten Ausgangszustand zu überführen.
- **Ausführung der Zustandslogik:** Sofern keine Fehlerbehandlung aktiv ist, wird die durch einen Funktionszeiger referenzierte Zustandsfunktion aufgerufen. Dies ermöglicht die schrittweise Abarbeitung implementierter Zustände. Mehr dazu in Kapitel 3.3.

Zu Beginn wird noch das TBIFG-Interrupt-Flag im TBOCTL-Register gelöscht, um die ISR für den nächsten Zyklus vorzubereiten. Die periodische Natur dieser Routine ist essentiell für weitere Funktionen wie die Timeout-Erkennung bei der Befehlseingabe und die nicht-blockierende Ausführung längerer Operationen.

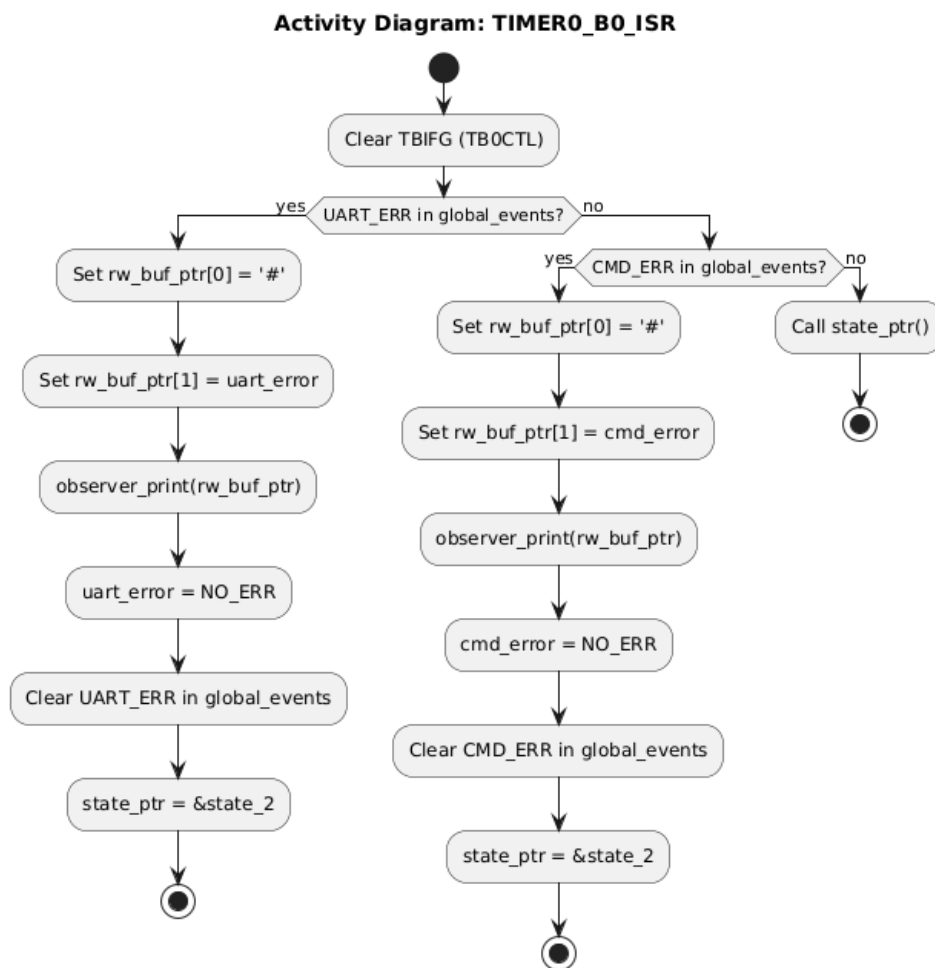


Abb. 3.3.: Aktivitätsdiagramm – Timer ISR

### 3.1.4. UART ISR

Eine weitere unerlässliche Komponente des Observer-Moduls ist die Interrupt Service Routine der UART-Kommunikation. Diese steuert den bidirektionalen Datenaustausch, d. h. das Senden und Empfangen von Zeichen über die serielle Schnittstelle. Sie behandelt zudem Grenzfälle, wie den Empfang ungültiger Zeichen oder spezifische UART-Fehlerzustände. Im Unterschied zur zyklisch operierenden Timer-ISR (siehe Kapitel 3.1.3) wird die UART-ISR ereignisgesteuert ausgelöst: entweder nach Empfang eines Datenbytes im Empfangspuffer (UCA0RXBUF) oder wenn der Sendepuffer (UCA0TXBUF) bereit ist, ein neues Datenbyte für die Übertragung aufzunehmen. Die Unterscheidung zwischen den verschiedenen UART-Ereignissen – wie z. B. Empfangen, Senden oder Fehler erkannt – erfolgt über das Interrupt-Vektor-Register UCA0IV. Tiefgreifendere Informationen darüber in Kapitel 2.2.3.2 und Kapitel 2.2.5.

Die Empfangslogik – definiert über den Interruptvektor USCI\_UART\_UCRXIFG – der UART-ISR lässt sich anhand des Aktivitätsdiagramms in Abbildung 3.4 präzise nachvollziehen. Zu den Kernaufgaben des Empfangsteils zählen:

- **Fehlerdetektion:** Prüfung auf Kommunikationsfehler wie UCBRK (Break-Signal) oder UCRXERR (Framing-, Overrun- oder Paritätsfehler) und entsprechendes Setzen eines UART-Fehler-Wertes – in der `uart_error`-Variable.
- **Zeichenbehandlung:** Verarbeitung empfangener Bytes – und zwischenspeichern in der `rx_byte`-Variable –, inklusive spezieller Steuerzeichen wie „\r“ (Carriage Return), welches die Befehlseingabe abschließt und die Zustandsmaschine aktiviert, sowie „\b“ (Backspace) zur Korrektur von bereits gespeicherten Eingaben im `uart_buffer`, und „\n“ (Newline), das ignoriert wird.
- **Pufferverwaltung:** Validierung (Prüfung auf Alphanumerik und Leerzeichen), Pufferung der gültigen Zeichen im `uart_buffer` inklusive Überlaufschutz und Echo-Ausgabe des Zeichens.
- **Timeout-Initiierung:** Bei Empfang des ersten Zeichens nach einem Reset wird `state_ptr` auf `state_0` gesetzt, welches die Timeout-Überwachung startet.

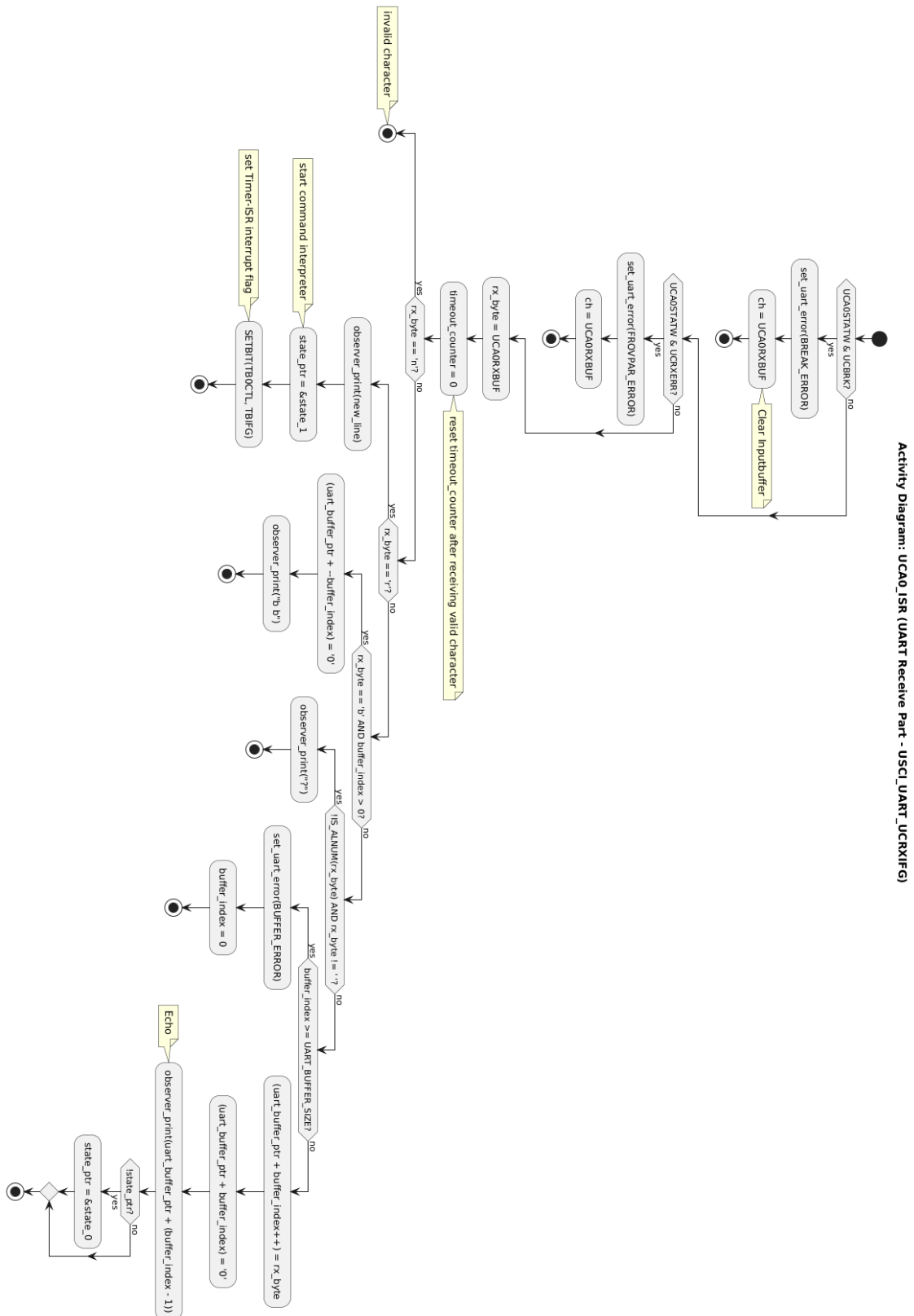


Abb. 3.4.: Aktivitätsdiagramm – Timer ISR

Der Sendeteil – mit dem Interruptvektor `USCI_UART_UCTXIFG` – der UART-ISR, dessen Ablauf in Abbildung 3.5 visualisiert ist, ist für die schrittweise Ausgabe von Zeichenketten zuständig. Solange `print_ptr` nicht auf ein Null-Terminierungszeichen (`'\0'`) zeigt, wird das aktuelle Zeichen in den Sendepuffer `UCA0TXBUF` geladen. Nach erfolgreicher Übertragung aller Zeichen einer Nachricht wird das `WRT_UART`-Flag in `global_events` zurückgesetzt und der Sendeinterrupt (`UCTXIE`) deaktiviert, während der Empfangsinterrupt (`UCRXIE`) reaktiviert wird. Diese ISR ist somit die direkte Schnittstelle für alle von `observer_print` initiierten Schreibvorgänge.

Activity Diagram: UCA0\_ISR (UART Send Part - USCI\_UART\_UCTXIFG)

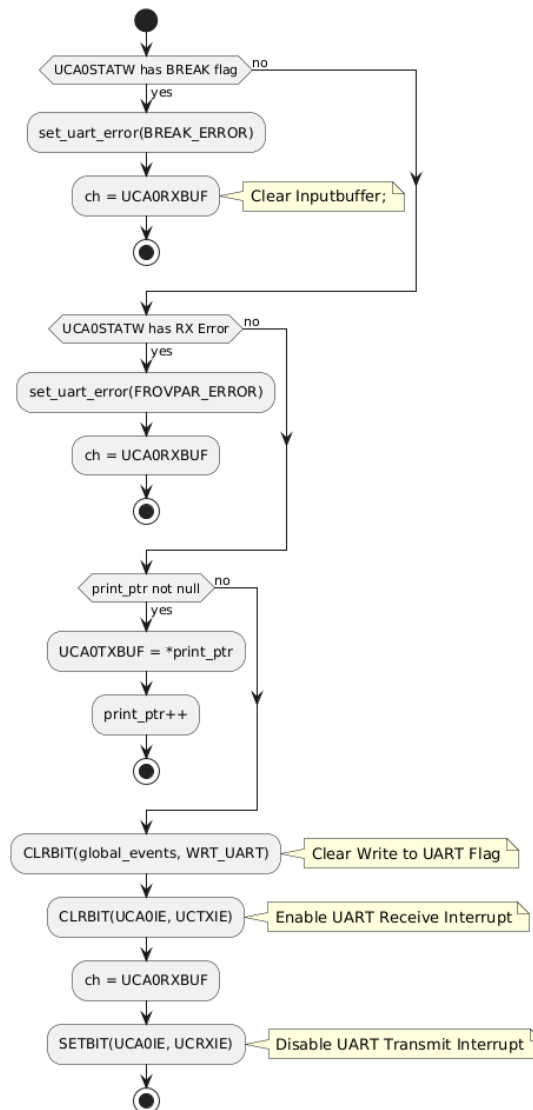


Abb. 3.5.: Aktivitätsdiagramm – Timer ISR

## 3.2. Zustandsmaschine und Hilfsstrukturen

Dieser Abschnitt beschreibt das „Rückgrat“ des Observer-Moduls. Dazu zählen die Zustandsmaschine sowie verschiedene unterstützende Funktionen und Datenstrukturen.

### 3.2.1. Die Zustandsmaschine

Das in Kapitel 3.1.1 erläuterte Konzept der zustandsbasierten Verarbeitung eingehender Befehle basiert auf den in Listing 3.1 dargestellten *Funktionsprototypen*<sup>37</sup>.

```

1  /*
2  * State Machine
3  */
4  LOCAL Void (*state_ptr)(Void);
5  LOCAL Void state_dummy(Void);
6  LOCAL Void state_0(Void);
7  LOCAL Void state_1(Void);
8  LOCAL Void state_2(Void);

```

**Listing 3.1:** Funktionszeiger für die Zustandsmaschine

Die angegebenen Funktionsprototypen repräsentieren die jeweiligen Zustände, in denen sich das System zu einem bestimmten Zeitpunkt befindet. Abhängig vom aktuellen Verarbeitungsschritt referenziert der zentrale Funktionszeiger `state_ptr` eine dieser Funktionen, die anschließend innerhalb der Timer-Interrupt-Service-Routine (ISR) ausgeführt wird.

- **state\_dummy** stellt den Leerlaufzustand des Systems dar, in dem keine Berechnungen stattfinden (Listing 3.2).

```

1  LOCAL Void state_dummy(Void) {}

```

**Listing 3.2:** Implementierung – state\_dummy

<sup>37</sup>Ein Funktionsprototyp legt Signatur und Rückgabetyt einer Funktion fest, ohne deren Implementierung bereitzustellen.

- In **state\_0** ist der Timeout-Timer für die Eingabe von Zeichen über die Benutzerkonsole aktiv (Listing 3.3).

```
1     LOCAL Void state_0(Void) {
2         timeout_counter++;
3         // Timeout Handling
4         if (timeout_counter >= TIMEOUT_THRESHOLD) {
5             timeout_counter = 0; // Reset counter
6             set_uart_error(TIME_OUT);
7             state_ptr = &state_2;
8         }
9     }
```

Listing 3.3: Implementierung – state\_0

- **state\_1** überprüft den Eingabepuffer auf das Vorliegen eines gültigen Befehls (Listing 3.4).

```
1     LOCAL Void state_1(Void) {
2         //SETBIT(P3OUT, BIT4);
3         if (*uart_buffer_ptr EQ '\0') {
4             state_ptr = &state_2;
5             //CLRBIT(P3OUT, BIT4);
6             return;
7         }
8         // Reset dict_ptr and set error
9         if (Observer_func_dict[dict_idx].func EQ NULL) {
10             dict_idx = 0;
11             set_cmd_error(UNKNOWN_CMD);
12             //CLRBIT(P3OUT, BIT4);
13             return;
14         }
15         // Compare buffer with dict entry
16         if (strcmp(uart_buffer, Observer_func_dict[dict_idx].key, 3) ==
17             0) {
18             // Extract functionpointer
19             state_ptr = Observer_func_dict[dict_idx].func;
20             return;
21         }
22         dict_idx++;
23         //CLRBIT(P3OUT, BIT4);
24     }
```

Listing 3.4: Implementierung – state\_1



- **state\_2** setzt das Observer-Modul in einen definierten Anfangszustand zurück (Listing 3.5).

```

1      LOCAL Void state_2(Void) {
2
3          //SETBIT(P3OUT, BIT4);
4          if (!TSTBIT(global_events & WRT_UART, WRT_UART)) {
5              mem_addr_ptr = 0;
6
7              blocks = 0;
8              mem_addr_idx = 0;
9              *rw_buf_ptr = ' ';
10             *(rw_buf_ptr + 1) = '\0';
11
12             buffer_index = 0;
13             *uart_buffer_ptr = '\0';
14             dict_idx = 0;
15
16             state_ptr = &state_dummy;
17         }
18         //CLRBIT(P3OUT, BIT4);
19     }

```

Listing 3.5: Implementierung – state\_2

Das in Kapitel 3.1.2 vorgestellte Dictionary, in dem Befehle ihren zugehörigen Funktionszeigern zugewiesen werden, implementiert Listing 3.6.

```

1      typedef struct {
2          const Char key[CMD_LENGTH]; // String key for the function
3          Void (*func)(Void);         // Pointer to the function
4      } ObserverFuncEntry;
5
6      LOCAL const ObserverFuncEntry Observer_func_dict[OBS_FUNC_CMD] = {
7          {"rdm", read_mem},           // read_mem function
8          {"wrm", write_mem},          // write_mem function
9          {"", NULL}
10     };

```

Listing 3.6: Dictionary für Befehls-Funktionszeiger verknüpfung

### 3.2.2. Zeichenweise Ausgabe

Die in Kapitel 3.1.3 erwähnte `observer_print`-Funktion dient der Initialisierung eines asynchronen Zeichenkettenausgabevorgangs über eine UART-Schnittstelle. Zunächst überprüft sie, ob der übergebene String-Zeiger `null` ist, was auf einen ungültigen Eingabewert hinweist; in diesem Fall wird ein Fehlerstatus gesetzt und der Vorgang abgebrochen. Andernfalls wird eine globale Ereignis-Flagge (`WRT_UART`) gesetzt, das die UART-Übertragung signalisiert. Der Zeiger auf die zu übertragende Zeichenkette wird in die globale `print_ptr`-Variable geschrieben, woraufhin durch Setzen der Interrupt-Flags `UCTXIFG` und `UCTXIE` der UART-Sendeinterrupt aktiviert wird. Dadurch wird der eigentliche Übertragungsvorgang vom Interrupt-Handler<sup>38</sup> durchgeführt, was eine nicht-blockierende Datenübertragung ermöglicht.

```
1  #pragma FUNC_ALWAYS_INLINE(observer_print)
2  LOCAL Void observer_print(const char * str) {
3      if (str EQ NULL) {
4          set_uart_error(PRINT_ERROR);
5          return;
6      }
7      SETBIT(global_events, WRT_UART);
8      print_ptr = str;
9      SETBIT(UCA0IFG, UCTXIFG); // UART Transmit Interrupt Flag
10     SETBIT(UCA0IE, UCTXIE);  // Enable UART Receive Interrupt
11     return;
12 }
```

**Listing 3.7:** Initiierungs-Funktion für UART Datenübertragung

<sup>38</sup>Ein Handler ist in der Informatik eine Routine oder Funktion, die auf ein bestimmtes Ereignis oder eine Bedingung reagiert, z. B. ein Ereignis- oder Interrupt-Handler.

### 3.3. Interruptgesteuertes Lesen und Schreiben

Aufbauend auf dem Grundkonzept des Observer-Moduls und der etablierten Architektur lassen sich weitere Funktionen und Features integrieren. Im Rahmen dieser Arbeit fiel die Wahl für eine Kernfunktion auf das interruptgesteuerte Lesen und Schreiben von FRAM-Speicherzellen. Die nachfolgenden Abschnitte erarbeiten die Konzeption und Implementierung dieser Speicheroperationen und analysieren potenzielle Limitationen und Hindernisse wie eine aktive MPU. Ein fundamentales Merkmal hierbei ist, dass die interruptgesteuerten Speicherzugriffe nicht zwingend einen expliziten Haltezustand des Gesamtsystems erfordern — wie im Kontext der möglichen Debugging-Funktion in Kapitel 3 erläutert. Die angestrebte Funktionalität kann auch ohne diese Art an erweiterter Systemkontrolle, eigenständig agieren. Ein potenzieller Nachteil besteht jedoch in der Konsistenz gelesener und geschriebener Daten: Durch die zustandsorientierte, inkrementelle Verarbeitung der Lese- und Schreibvorgänge könnten konkurrierende Prozesse den adressierten Speicherbereich zwischen einzelnen Verarbeitungsschritten modifizieren. Daraus ergibt sich ein weiterer Grund für die Notwendigkeit, das Grundkonzept des statusorientierten, interruptgesteuerten Betriebs umzusetzen – wie im Folgenden beschrieben.

#### 3.3.1. Grundkonzept des statusorientierten, interruptgesteuerten Betriebs

Die zustandsorientierte Programmierung für das Observer-Modul setzt die Dekomposition komplexer, sequenzieller Abläufe in kleinere, Verarbeitungseinheiten voraus. Wie bereits im Kapitel 3.1.1 und Kapitel 3.1.3 dargelegt, repräsentieren die Lese- und Schreiboperationen dedizierte Zustände innerhalb der übergeordneten Zustandsmaschine. Diese Unterteilung ist essentiell, um die Reaktionsfähigkeit des Systems auf externe Eingaben und die „parallele“ Bearbeitung anderer systeminterner Aufgaben sicherzustellen. Für die Realisierung der Speicherzugriffe sind neben generischen Zuständen – zur Befehlsverarbeitung und Systeminitialisierung – spezifische Zustände für das Lesen und Schreiben erforderlich. Da diese Operationen, insbesondere bei größeren Speicherzugriffen, potenziell zeitintensiv sind, ist die Überführung in eine zeichenorientierte Verarbeitung, welche periodisch durch die Timer-ISR ausgeführt wird, notwendig.

Konkret bedeutet dies, dass nach der Verarbeitung eines einzelnen Zeichens die jeweilige Speicheroutine temporär terminiert und die Kontrolle an das System zurückgibt. Die Fortsetzung der Operation erfolgt erst bei der nächsten Periode der Timer-ISR. Dieser inkrementelle Ansatz steigert die Reaktivität des Gesamtsystems erheblich. Im Fazit unter Kapitel 4.1.1 erfolgen hierzu Laufzeitmessungen unter Realbedingungen und werden genauer untersucht. Dem gegenüber steht der Nachteil, dass eine zusätzliche Laufvariable zum verfolgen des aktuellen Fortschritts, innerhalb der Lese- oder Schreiboperation, benötigt wird. Dadurch entsteht eine potentiell längere Gesamtdauer für die vollständige Operation, da pro Timer-Zyklus lediglich ein einzelnes Zeichen verarbeitet wird.<sup>39</sup>

### 3.3.2. Rolle der Interrupt Service Routinen

Die beiden Interrupt Service Routinen des Moduls, die UART-ISR und die Timer-ISR, erfüllen dabei kritische Funktionen im gesamten Prozesszyklus der Speicheroperationen — von der initialen Befehlseingabe über die schrittweise Datenverarbeitung bis hin zur finalen Ergebnisausgabe.

- **Die UART-ISR** agiert als primäre Schnittstelle für die Benutzerinteraktion. Sie ist verantwortlich für den Empfang vollständiger Befehlsstrings, wie beispielsweise „`rdm <adresse> <bytes>`“, zur Anforderung eines Speicherlesevorgangs oder „`wrm <adresse> <string>`“, zur Initiierung eines Schreibvorgangs. Darüber hinaus übernimmt die UART-ISR die Ausgabe der aus dem FRAM gelesenen Daten an den Benutzer und quittiert einen erfolgreich abgeschlossenen Schreibvorgang durch erneute Ausgabe des Befehls. Tiefgreifender in Kapitel 3.1.4 beschrieben.
- **Die Timer-ISR** fungiert als periodischer Auslöser und Taktgeber für die Ausführung zustandsbasierter Funktionen. Sie ruft wiederholt die im Funktionszeiger `state_ptr` referenzierte, aktuell aktive Zustandsfunktion (z. B. `read_mem` oder `write_mem`) auf. Diese Funktion bleibt so lange aktiv, bis ein definierter Endzustand der jeweiligen Operation erreicht ist. Weitere Informationen darüber in Kapitel 3.1.3.

---

<sup>39</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

### 3.3.3. Implementierung der Lese- und Schreibfunktion

Die softwaretechnische Realisierung der Lese- und Schreibfunktionen ist bewusst ressourcenschonend und auf Laufzeit optimiert. Diese Funktionen besitzen im Wesentlichen einen definierten Startzustand, einen Endzustand sowie die Logik für den eigentlichen, inkrementellen Speicherzugriff.

Der Programmablauf der `read_mem`-Funktion ist wie folgt strukturiert (siehe auch Quellcodeausschnitt in Listing 3.8):

- **Initiierung:** Die Funktion wird aktiv, sobald die Befehlsverarbeitungsroutine (`state_1`) einen „rdm“-Befehl identifiziert und den globalen Zustandszeiger `state_ptr` auf die Adresse der `read_mem`-Funktion setzt. Mit dem ersten Durchgang werden die für den Lesevorgang notwendigen Parameter — die Startadresse (gespeichert in `mem_addr_ptr`) und die Anzahl der zu lesenden Bytes (`blocks`) — aus dem `uart_buffer` extrahiert und in globalen Variablen zwischengespeichert.
- **Inkrementelle Ausführung:** Bei jedem Aufruf durch die Timer-ISR wird exakt ein Byte aus dem FRAM an der aktuellen Adresse (`mem_addr_ptr + mem_addr_idx`) gelesen. Der Fortschritt innerhalb des adressierten Speicherbereichs wird mittels der Indexvariable `mem_addr_idx` verfolgt, welche nach jedem gelesenen Byte inkrementiert wird.
- **Datenausgabe:** Das eingelesene Zeichen wird einer Validierung unterzogen (Filterung nicht-alphanumerischer Zeichen, die durch Leerzeichen ersetzt werden, zu sehen in Listing 3.10) und anschließend im Puffer `rw_buf` abgelegt. Mittels der Zuweisung „`print_ptr = rw_buf_ptr`“ und setzen der Bits für das senden von Daten über das UART-Protokoll erfolgt die unmittelbare Ausgabe des gelesenen Zeichens an die Benutzerkonsole.
- **Beendigung:** Die Leseoperation gilt als abgeschlossen, sobald der Zähler `mem_addr_idx` die vorgegebene Anzahl an zu lesenden Bytes erreicht oder überschritten hat. In diesem Fall wird der `state_ptr` auf den Zeiger der `state_2`-Funktion (Rücksetz- und Leerlaufzustand) umgelenkt, um das System für neue Befehle vorzubereiten.

```
1  #pragma FUNC_ALWAYS_INLINE(read_mem)
2  LOCAL Void read_mem(Void) {
3
4      //SETBIT(P3OUT, BIT4);
5      if (TSTBIT(global_events & WRT_UART, WRT_UART)) {
6          //CLRBIT(P3OUT, BIT4);
7          return;
8      }
9
10     if (mem_addr_idx > (blocks - 1)) {
11         state_ptr = &state_2;
12         //CLRBIT(P3OUT, BIT4);
13         return;
14     }
15
16     if (mem_addr_ptr EQ NULL) {
17         // Extract useful arguments
18         Char *mem_addr_str = strtok(uart_buffer_ptr + 4, " ");
19         Char *block_str = mem_addr_str + strlen(mem_addr_str) + 1;
20
21         mem_addr_ptr = (Char *)strtol(mem_addr_str, NULL, 0);
22         blocks = (UInt)atoi(block_str);
23     }
24
25     *rw_buf_ptr = *((volatile Char *)mem_addr_ptr + mem_addr_idx);
26     if (!IS_ALNUM(*rw_buf_ptr)) {
27         *rw_buf_ptr = ' ';
28     }
29
30     print_ptr = rw_buf_ptr;
31     SETBIT(global_events, WRT_UART);
32     SETBIT(UCA0IFG, UCTXIFG); // UART Transmit Interrupt Flag
33     SETBIT(UCA0IE, UCTXIE);  // Enable UART Receive Interrupt
34
35     mem_addr_idx++;
36
37     //CLRBIT(P3OUT, BIT4);
38 }
```

**Listing 3.8:** Implementierungsausschnitt der read\_mem-Funktion im Observer-Modul

Analog dazu ist der Programmablauf der `write_mem`-Funktion konzipiert, welcher eine inaktive MPU voraussetzt: (siehe Quellcodeausschnitt in Listing 3.9)

- **Initiierung:** Die Aktivierung erfolgt, wenn die Befehlsverarbeitung (`state_1`) einen „`wrm`“-Befehl erkennt und `state_ptr` auf die `write_mem`-Funktion setzt. Die Zieladresse (`mem_addr_ptr`) und der zu schreibenden String (`write_str_ptr`) werden aus dem `uart_buffer` extrahiert.
- **Adressvalidierung:** Vor dem eigentlichen Schreibzugriff wird überprüft, ob die angegebene Zieladresse in potenziell schreibgeschützten oder für andere Zwecke reservierten Speicherbereichen liegt (z. B. im Hexadezimalbereich von 0x000000 bis 0x0001FF oder 0x000C00 bis 0x000FFF, mehr Informationen dazu in Abbildung 3.6). Bei Detektion einer ungültigen Adresse wird ein entsprechender Fehlercode (`INV_ADDR`) gesetzt und die Operation terminiert.

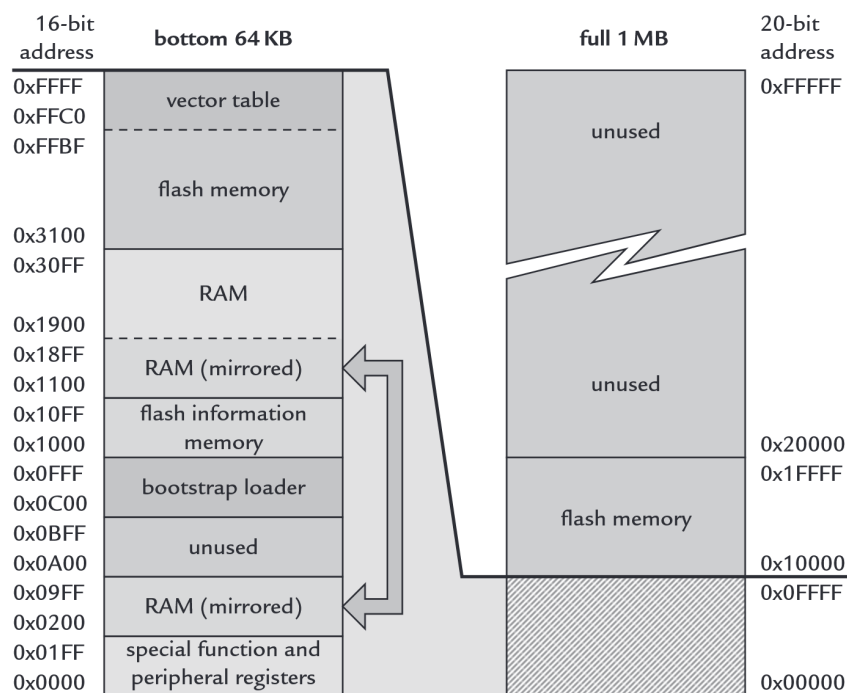


Abb. 3.6.: Memory map des FG4618 [4, S. 603, Fig. 11.1]

- **Inkrementelle Ausführung:** Bei jedem Aufruf durch die Timer-ISR wird ein einzelnes Zeichen aus dem `write_str_ptr` (an der durch `mem_addr_idx` indizierten Position) in die entsprechende FRAM-Speicherzelle (`mem_addr_ptr + mem_addr_idx`) geschrieben. Der Fortschritt wird durch das hochzählen der `mem_addr_idx`-Variable verfolgt.

- **Datenausgabe/Echo:** Das soeben geschriebene Zeichen wird zur Verifikation in den `rw_buf` kopiert und mittels `observer_print` als Echo an den Benutzer zurückgesendet.
- **Beendigung:** Die Schreiboperation ist abgeschlossen, sobald das Null-Terminierungszeichen des `write_str_ptr` erreicht wurde. Daraufhin wird abermals der `state_ptr` auf die `state_2`-Funktion gesetzt, um den Schreibzyklus ordnungsgemäß zu beenden und das Modul in seinen definierten Ausgangszustand zurück zu setzen.

```
1  #pragma FUNC_ALWAYS_INLINE(write_mem)
2  LOCAL Void write_mem(Void) {
3
4      //SETBIT(P3OUT, BIT4);
5      if (TSTBIT(global_events & WRT_UART, WRT_UART)) {
6          //CLRBIT(P3OUT, BIT4);
7          return;
8      }
9
10     if (mem_addr_ptr EQ NULL) {
11         // Extract useful arguments
12         Char *mem_addr_str = strtok(uart_buffer_ptr + 4, " ");
13         write_str_ptr = mem_addr_str + strlen(mem_addr_str) + 1;
14
15         mem_addr_ptr = (Char *)strtol(mem_addr_str, NULL, 0);
16     }
17
18     if (between("0x000000", mem_addr_ptr, "0x0001FF")
19         || between("0x000C00", mem_addr_ptr, "0x000FFF")) {
20         set_cmd_error(INV_ADDR);
21         return;
22     }
23
24     *rw_buf_ptr = *((volatile Char *)write_str_ptr + mem_addr_idx);
25
26     if (*rw_buf_ptr EQ '\0') {
27         state_ptr = &state_2;
28         //CLRBIT(P3OUT, BIT4);
29         return;
30     }
31
32     if (!IS_ALNUM(*rw_buf_ptr)) {
33         *rw_buf_ptr = ' ';
```



```

34     }
35
36     print_ptr = rw_buf_ptr;
37     SETBIT(global_events, WRT_UART);
38     SETBIT(UCA0IFG, UCTXIFG); // UART Transmit Interrupt Flag
39     SETBIT(UCA0IE, UCTXIE);  // Enable UART Receive Interrupt
40
41     *((volatile Char *)mem_addr_ptr + mem_addr_idx) = *rw_buf_ptr;
42     mem_addr_idx++;
43
44     //CLRBIT(P3OUT, BIT4);
45 }

```

**Listing 3.9:** Implementierungsausschnitt der `write_mem`-Funktion im Observer-Modul

Kurzer Einschub zur Prüfung auf Alphanumerische Zeichen mittels eines eigenen *Makros*<sup>40</sup>. Findet auch Verwendung in der zuvor, in Kapitel 3.1.4, besprochenen UART-ISR.

```

1  /*
2  * Custom Makros
3  */
4  #define IS_ALNUM(ch) (between('0', (ch), '9') ||
5                       between('A', (ch), 'Z') ||
6                       between('a', (ch), 'z'))

```

**Listing 3.10:** Zeichenprüfung auf Alphanumerik mittels eigen Makros

Die Identifikation, Verarbeitung und Signalisierung potenziell auftretender Fehlerzustände ist ein weiterer unerlässlicher Entwicklungsschritt. Dieser Aspekt der Fehlerbehandlung wird im nachfolgenden Kapitel eingehend thematisiert.<sup>41</sup>

<sup>40</sup>Ein Makro ist eine Anweisung oder ein Symbol, das durch eine andere Zeichenfolge ersetzt wird, typischerweise zur Automatisierung wiederkehrender Codeabschnitte oder Befehle.

<sup>41</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

## 3.4. Fehlerbehandlung

Die Identifizierung und Auswertung von Randbedingungen sowie die daran anschließende Fehlerbehandlung sind grundlegende Bestandteile komplexer Softwareprojekte. Für den Nutzer ist die Ausgabe potenziell auftretender Fehlerzustände von hoher Bedeutung, um eine angemessene Fehlerbehebung einleiten zu können. Um eine Überlastung des Nutzers mit redundanten oder niederrangigen Fehlermeldungen – oder Folgefehlern – zu vermeiden, ist eine interne Speicherung und anschließende Priorisierung detektierter Fehler notwendig.

Die primären Fehlerquellen innerhalb des Observer-Moduls ist die UART- Kommunikationsschnittstelle sowie die Prozesse der Befehlsinterpretation und -ausführung.

- **Die UART-Schnittstelle** besitzt diverse Fehlerindikatoren (Fehlervektoren beschrieben in Kapitel 2.2.4), welche direkt zu Beginn der UART-ISR – sowohl im Sende- als auch im Empfangsteil – geprüft werden. Ein zusätzlicher Fehlerfall ist die überschreitung eines definierten Zeitlimits (Time-Out) pro Zeichen für die Befehlseingabe. Bei Erreichen dieses Schwellenwerts wird die laufende Befehlseingabe terminiert, das System auf einen definierten Ausgangszustand zurückgesetzt und eine entsprechende Fehlermeldung an die verbundene Benutzerkonsole übermittelt.
- **Die Befehlsinterpretation sowie -verarbeitung** kann ebenfalls Fehler und definierte Randbedingungen aufweisen. Hierzu zählt die Detektion unbekannter Befehle, die Verwendung invalider Speicheradressen (z. B. Zugriff auf geschützte Bereiche, Vgl. Abbildung 3.6), fehlerhafte Zeichenketten oder unzulässige Blockgrößen.

Die hierfür implementierten Funktionen sind in Listing 3.11 dargestellt. Unterschiedliche Fehlerquellen erfordern dabei eine separate Behandlung und Klassifizierung.

```
1  #pragma FUNC_ALWAYS_INLINE(set_uart_error)
2  LOCAL Void set_uart_error(Char err) {
3      if (err == NO_ERR || uart_error < err) {
4          uart_error = err;
5          SETBIT(global_events, UART_ERR);
6      }
7  }
8
9  #pragma FUNC_ALWAYS_INLINE(set_cmd_error)
10 LOCAL Void set_cmd_error(Char err) {
11     if (err == NO_ERR || cmd_error < err) {
12         cmd_error = err;
13         SETBIT(global_events, CMD_ERR);
14     }
15 }
```

**Listing 3.11:** Funktion zur Fehlerpriorisierung und Speicherung sowie Setzen eines globalen Error-Flags

Die Fehlerpriorisierung basiert auf in der Header-Datei (`Observer.h`) definierten Makros, die entsprechend ihrer Eskalationsstufe numerisch kodiert sind (Listing 3.12). Mittels eines Vergleichsoperators wird evaluiert, ob ein neu aufgetretener Fehler eine höhere Priorität besitzt als ein bereits intern vermerkter Fehler. Nur wenn dies der Fall ist oder noch kein spezifischer Fehler dieser Kategorie gespeichert wurde, wird der neue Fehler als der aktuell relevanteste behandelt und der bisherige gegebenenfalls überschrieben.

```
1  /*
2  * UART Error
3  */
4  #define NO_ERR          0x00    // no error
5  #define TIME_OUT       'A'     // time out
6  #define BUFFER_ERROR   'B'     // buffer error (e.g. to many bytes received)
7  #define FROVPAR_ERROR  'C'     // frame overrun or parity error
8  #define BREAK_ERROR    'D'     // break error (lost communication)
9  #define PRINT_ERROR    'E'     // unable to print on UART
10
11 /*
12 * Command Error
13 */
14 #define UNKNOWN_CMD     '1'     // unknown command
15 #define INV_PTR         '2'     // Invalid function pointer
16 #define INV_ADDR        '3'     // Invalid memory address
17 #define INV_BLK         '4'     // Invalid block-size
18 #define INV_STR         '5'     // Invalid string
```

**Listing 3.12:** Definition der Fehler-Makros in der Header-Datei `Observer.h` zur Festlegung der Fehlerprioritäten.

Die Konstruktion und Ausgabe der finalen Fehlermeldung obliegt der Timer-ISR. Diese Routine identifiziert den spezifischen Fehlerkontext anhand der zuvor gesetzten Fehlerereignis-Flag im globalen Ereignisregister `global_events`. Der zugehörige Fehlercode im Fehlerregister `uart_error` oder `cmd_error`, zusammen mit einem Präfix-Zeichen („#“) ergibt anschließend die Fehlermeldung. Zur Formatierung wird der Lese- und Schreib-Puffer `rw_buf` genutzt. Dessen Größe ist auf drei Byte initialisiert, um das Präfix-Zeichen, den alphanumerischen Fehlercode-Wert und das Null-Terminierungszeichen aufzunehmen. Die Startadresse dieses Puffers wird im Anschluss der Funktion `observer_print` übergeben, welche die zeichenweise Übertragung der Fehlermeldung via UART initiiert. Es ist hierbei zu beachten, dass die Timer-ISR darauf ausgelegt ist, pro Durchlauf einen Fehler zu behandeln, da eine weitere Fehlerbehandlung die zuvor formatierte Fehlermeldung überschreiben würde. Nach der Ausgabe eines Fehlers wird der Systemzustand abermals auf einen definierten Ausgangszustand zurückgesetzt. Die hier beschriebene Routine zur Fehlerbehandlung in der Timer-ISR ist in Listing 3.13 visualisiert.

```
1  /*
2  * Error Handling
3  */
4  if (TSTBIT(global_events & UART_ERR, UART_ERR)) {
5      *rw_buf_ptr = '#';
6      *(rw_buf_ptr + 1) = uart_error;
7      observer_print(rw_buf_ptr);
8      uart_error = NO_ERR;
9      CLRBIT(global_events, UART_ERR);
10     state_ptr = &state_2;
11     return;
12 }
13
14 if (TSTBIT(global_events & CMD_ERR, CMD_ERR)) {
15     *rw_buf_ptr = '#';
16     *(rw_buf_ptr + 1) = cmd_error;
17     observer_print(rw_buf_ptr);
18     cmd_error = NO_ERR;
19     CLRBIT(global_events, CMD_ERR);
20     state_ptr = &state_2;
21     return;
22 }
```

**Listing 3.13:** Zusammensetzung und Initialisierung der Fehlernachrichtausgabe

Nachdem Mechanismen zur Fehleridentifikation und -ausgabe etabliert wurden, richtet sich der Fokus nun auf die Realisierung eines erweiterten Funktionsmerkmals. Die folgenden Kapitel widmen sich daher der Analyse verschiedener Debugging-Methoden und untersuchen im Speziellen den Einsatz von Software-Breakpoints im Kontext eingebetteter Systeme. Dabei steht insbesondere die Evaluation einer Implementierungsmöglichkeit und den Mehrwert von Software-Breakpoints für das Observer-Modul im Vordergrund, speziell im Hinblick auf die funktionale Ergänzung bereits bestehender Funktionen (Mit Referenz auf den in Kapitel 3 erwähnten Sachverhalt).<sup>42</sup>

<sup>42</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

## 3.5. Debugging-Methoden: Hardware- vs. Software-Breakpoints

Im Kontext der Fehlersuche und Programmanalyse in Eingebetteter Software stellen Breakpoints ein fundamentales Werkzeug dar. Sie ermöglichen es, die Ausführung eines Programms an einer vordefinierten Stelle zu unterbrechen, um den internen Zustand des Systems zu inspizieren. Grundsätzlich lassen sich zwei primäre Arten von Breakpoints unterscheiden: Software-Breakpoints und Hardware-Breakpoints, deren Implementierung und Eigenschaften sich signifikant unterscheiden.

Software-Breakpoints werden zur aktiven Laufzeit des Programms durch einen direkten Eingriff in den ausführbaren Code im Speicher des Mikrocontrollers realisiert. An der Zieladresse wird hierbei die ursprüngliche Programminstruktion temporär durch eine Breakpoint-Instruktion oder einen Trap-Befehl ersetzt, der einen Software-Interrupt oder eine Exception auslöst. Sobald der Programmzähler diese modifizierte Stelle erreicht, unterbricht der Mikrocontroller den normalen Programmfluss der Programmzähler stoppt und eine Debug-Routine wird ausgeführt. Der *Debugger*<sup>43</sup> kann diesen Zustand erkennen, die ursprüngliche Instruktion wiederherstellen und dem Entwickler die Kontrolle übergeben. Durch diesen Mechanismus sind Software-Breakpoints hochflexibel und können an nahezu jeder beliebigen Stelle im beschreibbaren Code-Speicher (wie RAM oder FRAM) gesetzt werden.

Die Vorteile des softwarebasierten Ansatzes liegen primär in der Möglichkeit, eine praktisch unbegrenzte Anzahl von Breakpoints im System zu verwenden, sowie in den geringen Anforderungen an zusätzliche, dedizierte Hardwarekomponenten auf dem Zielsystem selbst. Die grundlegende Fähigkeit, Interrupts oder Exceptions zu behandeln, ist hierfür ausreichend. [22, Kap. 4.7.16]

Gegenüber den Software-Breakpoints bieten hardwarebasierte Breakpoints den Vorteil, dass sie Programmunterbrechungen auch in solchen Speichersegmenten ermöglichen, die schreibgeschützt sind (z.B. *ROM* oder spezifisch geschützte Flash-Bereiche). Ein weiterer entscheidender Vorteil ist ihre Nicht-Intrusivität: Da keine Modifikation des Programmcodes stattfindet, werden weder die Konsistenz des Codes im Speicher noch das präzise Echtzeitverhalten (Timing) des Programms durch den Breakpoint-Mechanismus selbst beeinflusst. [22, 1, Vgl. Kap. 4.7.16, S. 54, Kap. 4.3]

---

<sup>43</sup>Ein Werkzeug zur schrittweisen Ausführung und Analyse von Programmen. Es erlaubt das Setzen von Haltepunkten, das Überprüfen von Speicherinhalten und das Nachvollziehen von Kontrollflüssen zur Fehlersuche und -behebung.

Um dies jedoch zu erreichen, benötigen Hardware-Breakpoints ein dediziertes Hardwaremodul innerhalb des Mikrocontrollers. Im Falle der MSP430-Mikrocontrollerfamilie ist dies das **Embedded Emulation Module (EEM)**. [11, S. 569, Kap. 21]

Dieses Modul beinhaltet spezielle Hardwareregister, typischerweise *Adresskomparatoren*<sup>44</sup>, welche die Speicheradresse des Befehls halten, an welcher bei Übereinstimmung mit dem Programmzähler ein Breakpoint ausgelöst werden soll. Da der Breakpoint durch externe Hardwarelogik ausgelöst wird und nicht durch eine im Programmablauf ausgeführte Instruktion, müssen seitens des Breakpoint-Mechanismus selbst keine Registerinhalte oder Stack-Elemente explizit zwischengespeichert und im Nachhinein wiederhergestellt werden. Ganz im Unterschied wie es bei einem durch einen Software-Breakpoint induzierten Interrupt der Fall sein kann. Die Zustandssicherung erfolgt erst durch die Debug-Routine nach erfolgter Unterbrechung. Der wesentliche Nachteil hierbei ist allerdings die strikte Limitierung der Anzahl gleichzeitig setzbarer Hardware-Breakpoints, welche direkt von der Anzahl der im EEM verfügbaren Komparator-Register abhängt. Für den MSP430 sind dies oft nur bis zu zwei. [11, 22, S. 570, Kap. 21.1, Vgl. Kap. 7.1]

Diese Gegenüberstellung offenbart einen klaren *Trade-off*<sup>45</sup>: Hardware-Breakpoints glänzen durch ihre Transparenz und die Fähigkeit, in geschützten Speicherbereichen zu operieren, sind jedoch eine knappe Ressource. Software-Breakpoints hingegen bieten eine hohe Flexibilität und nahezu unbegrenzte Verfügbarkeit, gehen aber mit einer leichten Modifikation des Programmcodes und potenziellen, wenn auch meist minimalen, Timing-Veränderungen einher. Angesichts der begrenzten Anzahl an Hardware-Breakpoints auf der MSP430-Plattform, die insbesondere bei komplexeren Debugging-Szenarien schnell erschöpft sein können, erweist sich die Implementierung von Software-Breakpoints als eine pragmatische und oft notwendige Erweiterung der Debugging-Möglichkeiten. Um die technischen Rahmenbedingungen für die Realisierung solcher Software-Breakpoints auf dem MSP430FR5729 sowie die Interaktion mit der Debugging-Infrastruktur genauer zu verstehen, ist eine detaillierte Betrachtung des eingesetzten Debug-Adapters und seiner Funktionsweise unerlässlich.

---

<sup>44</sup>Adresskomparatoren vergleichen Speicheradressen zweier Operanden, um deren Gleichheit oder relative Ordnung (größer/kleiner) festzustellen, typischerweise zur Steuerung von Sprüngen oder Zugriffsentscheidungen in Programmen.

<sup>45</sup>Abwägung zwischen zwei konkurrierenden Zielen, Konzepten, oder ähnlichem, bei der die Verbesserung des einen mit der Verschlechterung des anderen einhergeht.

Darüber hinaus existiert auch noch eine spezielle Art von Breakpoints welcher von einem Speicherzugriff ausglöst wird. Watchpoints können Grenzfälle identifizieren um dadurch invalide Speicheradressen und zugriffe sowie Pufferüberläufe zu analysieren. [22, Kap. 7.4.16.2]

Die nachfolgende Analyse des MSP-FET Debuggers wird weitere Aspekte des Software und Hardware-Basierten Debuggings beleuchten und die Grundlage für die spätere Implementierungsstrategie legen.<sup>46</sup>

### 3.5.1. Der MSP-FET Download Adapter im Detail

Die effektive Nutzung von sowohl Hardware- als auch Software-Breakpoints auf dem MSP430FR5729 ist maßgeblich von der externen Debugging-Hardware und -Software abhängig. Als zentrale Schnittstelle zwischen der Entwicklungsumgebung auf dem Host-PC und dem Ziel-Mikrocontroller dient in diesem ökosystem der **MSP-FET-Debugger (Flash Emulation Tool)**. Dieses externe Gerät, zu sehen in Abbildung 3.7, stellt die physische und logische Verbindung einem MSP430 her und ermöglicht tiefgreifende Eingriffe und Beobachtungen während der Programmausführung.

Der MSP-FET kommuniziert mit dem MSP430-Mikrocontroller typischerweise über standardisierte (Debug-)Schnittstellen wie *JTAG* (Joint Test Action Group) oder das von Texas Instruments entwickelte *Spy-Bi-Wire*<sup>47</sup> (*SBW*). Über diese Schnittstellen erhält der MSP-FET Zugriff auf das EEM des MSP430FR5729. Wie im vorherigen Kapitel 3.5 dargelegt, ist das EEM für die Realisierung von Hardware-Breakpoints zuständig. Als Vermittler überträgt der MSP-FET die vom Entwickler in der *IDE* (Code Composer Studio) definierten Hardware-Breakpoint-Adressen an die entsprechenden Register des EEM. Gleichzeitig empfängt er die vom EEM generierten Haltesignale und übermittelt diese an die IDE. Somit ist der MSP-FET unerlässlich für die Konfiguration und Nutzung der limitierten, aber präzisen Hardware-Breakpoint-Ressourcen des Mikrocontrollers. [4, S. 58, Kap. 3.4]

---

<sup>46</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

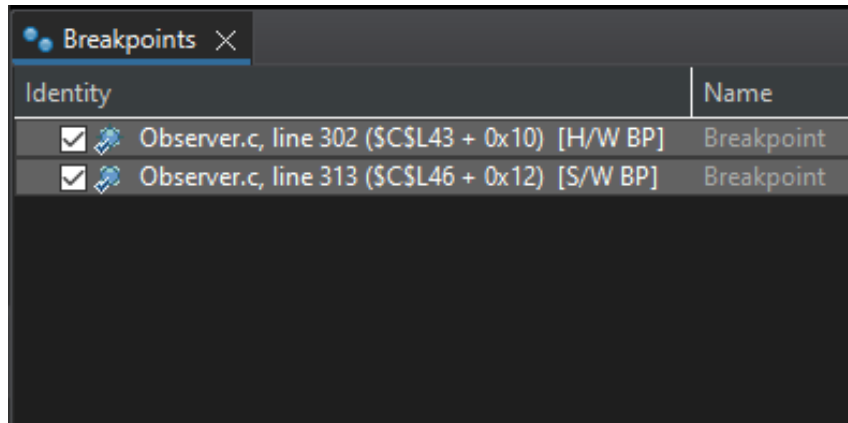
<sup>47</sup>Zweidraht-Variante des JTAG-Protokolls, die Pin-Anzahl am Target reduziert und besonders für platzkritische Anwendungen von Vorteil ist.





**Abb. 3.7.:** Flash Emulation Tool Programmer and Debugger [23]

Darüber hinaus spielt der MSP-FET eine ebenso große Rolle für die Implementierung und Handhabung von Software-Breakpoints. Die Fähigkeit, den Speicher des MSP430FR5729 (sowohl RAM als auch der beschreibbare Teil des FRAM) zur Laufzeit zu lesen und zu schreiben, ist die Grundvoraussetzung, um Instruktionen mit einer Software-Breakpoint-Routine zu ersetzen. Der Debugger liest über den MSP-FET die ursprüngliche Instruktion an der Zieladresse aus, ersetzt diese durch eine Breakpoint-Instruktion und, nach dem Auslösen des Software-Interrupts, stellt er die ursprüngliche Instruktion wieder her. Ferner ermöglicht der MSP-FET die Steuerung des Programmflusses (Anhalten, Starten, Einzelschrittbetrieb) und den Zugriff auf CPU-Register und Speicherinhalte, was für die Analyse des Systemzustands an einem Breakpoint unerlässlich ist. Das vom Software-Breakpoint ausgelöste Interrupt-Signal wird ebenfalls über die Debug-Schnittstelle an den MSP-FET und somit an die Host-Debugger-Software gemeldet. Wie solche Hardware und Software-Breakpoints in der Entwicklungsumgebung aussehen, ist in Abbildung 3.8 zu sehen. Das „H/W“ oder „S/W“ innerhalb der eckigen Klammern steht wahlweise für „Hardware“ oder „Software“ welches von einem „BP“ für „Breakpoint“ ergänzt wird.



**Abb. 3.8.:** Code Composer Studio - Breakpoint Übersicht

Es ist wichtig zu verstehen, dass der MSP-FET primär die Kommunikationsinfrastruktur und die Low-Level-Zugriffsmechanismen bereitstellt. Während er das Setzen von Hardware-Breakpoints direkt über das EEM steuert, stellt er für Software-Breakpoints die notwendigen Lese-, Schreib- und Kontrolloperationen zur Verfügung. Die eigentliche Logik eines Software-Breakpoints – das heißt, welche Instruktion als Breakpoint-Befehl dient, wie die ursprüngliche Instruktion gesichert und wiederhergestellt wird sowie der resultierende Trap behandelt wird – muss in der Debugger-Software auf dem Host und gegebenenfalls durch eine minimale Debug-Monitor-Routine auf dem *Target*<sup>48</sup> implementiert werden, wobei der MSP-FET als Brücke dient. [15, 13, S. 56, Kap. 10]

Die Kenntnis der Funktionalitäten und der Arbeitsweise des MSP-FET ist somit entscheidend für die Entwicklung einer robusten Software-Breakpoint-Lösung. Er definiert die Grenzen und Möglichkeiten, wie mit dem Target-System interagiert werden kann, um Breakpoints zu setzen, Zustandsinformationen abzufragen und die Programmausführung zu steuern. Die im Folgenden zu entwickelnde Strategie zur Implementierung von Software-Breakpoints muss sich daher eng an den durch den MSP-FET und die Debug-Schnittstelle des MSP430FR5729 gegebenen Rahmenbedingungen orientieren.

<sup>48</sup>Das Zielobjekt oder die Zielumgebung, auf das bzw. in die eine Operation, ein Befehl oder eine Ausführung abzielt – z. B. bei Compilern die Zielplattform, bei Build-Systemen das zu erstellende Artefakt.

### 3.5.2. Konzeptionierung von Software-Breakpoints

Zur Realisierung von Breakpoints existieren mehrere Ansätze. Ein bewährter Einstieg besteht darin, etablierte Debugger und ihre Architektur zu studieren. Im Embedded- und Low-Power-Bereich kommen beispielsweise Werkzeuge wie **TRACE32**, **M-Core** oder das zuvor besprochene MSP-FET von Texas Instruments zum Einsatz. Diese Debugger setzen Hardware-Breakpoints über spezielle Debug-Interfaces um und bieten damit eine hohe Zuverlässigkeit bei minimaler Eingriffstiefe in das Laufzeitsystem.

Im Gegensatz dazu zielt die hier vorgestellte Lösung auf eine Abwandlung der Software-Breakpoints ab, die direkt den Programmspeicher manipuliert. Dabei ersetzt das System, in dieser Implementierung, an der gewünschten Halteadresse den originalen Maschinenbefehl durch einen Sprungbefehl (Jump), der auf eine speziell implementierte **Breakpoint-Handler**-Routine verweist. Beim Erreichen dieses Befehls leitet das System zunächst einen kritischen Abschnitt ein: Es werden die für den Prozess relevanten Register (R1 bis R3) – namentlich *Program Counter*<sup>49</sup> (*PC*), *Stack Pointer*<sup>50</sup> (*SP*), *Statusregister*<sup>51</sup> (*SR*) und ggf. mehrere *General-Purpose-Register*<sup>52</sup> (R4 bis R15) – gesichert und Interrupts deaktiviert, um eine atomare Kontextsicherung zu gewährleisten. Anschließend wechselt das System in die Handler-Routine. Diese blockiert das gesamte System, mit Ausnahme des Observer-Moduls, um weiterhin das Auslesen und Manipulieren von Speicherinhalten zu ermöglichen. [11, 4, 3, S.91, Kap. 4.3, S. 31, Kap. 2.4]

Nach der Analyse stellt das System den ursprünglichen Programmzustand, durch das Laden der Registersätze, wieder her. Der Compiler stellt hierfür *Compiler-Intrinsics*<sup>53</sup> bereit wie `__get_SR`, `__get_SP` und `__set_interrupt_state`. Auf diese Weise wird ein vollständiger Zyklus von Unterbrechung, Inspektion und Fortsetzung des Programmflusses realisiert, ohne dass das Hauptprogramm von dem Observer-Modul tiefgreifender beeinflusst wird. [10, S.137, Kap. 6.8.1]

Vor diesem Hintergrund wird in den folgenden Abschnitten die Konzeptionierung erweiterter Software-Breakpoints im Detail erläutert und auf die dafür notwendigen Voraussetzungen eingegangen.

---

<sup>49</sup>Ein Register, das die Speicheradresse des derzeitigen Befehls enthält.

<sup>50</sup>Ein Register, das die Speicheradresse des letzten oder ersten Datenelements im Stack speichert.

<sup>51</sup>Register für eine Reihe von Flags, die von der arithmetisch-logischen Einheit in Abhängigkeit der zuletzt durchgeführten Rechenoperation gesetzt werden.

<sup>52</sup>Frei verwendbare Arbeitsregister für allgemeine Operationen.

<sup>53</sup>Compiler-spezifische, vordefinierte Funktionen als optimierter Assemblercode.

### 3.5.2.1. Verwendung eines Sprungbefehls anstelle einer Trap

Wie in Kapitel 3.5 erläutert, basieren klassische software-Breakpoints häufig auf sogenannten Trap-Instruktionen. Das Debugging-Framework der Entwicklungsumgebung unterstützt dies unter anderem über die **General Extension Language (GEL)** – ein Makro- und Skriptsystem, das die Initialisierung von Systemen sowie die Steuerung von Debug-Sitzungen ermöglicht. Darüber hinaus erlaubt GEL auch das Setzen von Breakpoints, Speicherzugriffe und die Ablaufkontrolle des Programms [22, Kap. 7.7, 7.7.8.6 & 7.7.8.7].

Für diese Arbeit war es jedoch erforderlich, dass bei Auslösen eines Breakpoints benutzerdefinierte Routinen aus dem Observer-Modul aktiv bleiben und verwendet werden können. Dazu gehören Funktionen, die in Kapitel 3.3 besprochen wurden. Diese Funktionalitäten sind mit dem Standardverhalten von GEL-Traps nicht kompatibel, da sie typischerweise nur auf Debugger-seitige Prozesse abzielen und keine Kontextintegration benutzerdefinierter Routinen auf dem Zielsystem erlauben.

Aus diesem Grund wurde bewusst auf die Verwendung eines Trap-Mechanismus verzichtet und stattdessen ein direkter Sprungbefehl auf eine benutzerdefinierte Breakpoint-Handler-Routine implementiert. Dadurch wird sichergestellt, dass das Observer-Modul auch während einer Unterbrechung aktiv bleibt und die Kontrolle über Register- und Speicherzugriffe erhalten bleibt.

### 3.5.2.2. Implementierung erweiterter Software Breakpoints

Die Grundidee von Software-Breakpoints besteht darin, an einer Stelle im Programmspeicher, an der ein gültiger Maschinenbefehl (*Opcod*<sup>54</sup>) liegt, diesen temporär durch einen Sprung auf die Breakpoint-Handler-Routine zu ersetzen. Zunächst wird der originale Opcode gesichert, um ihn später unverändert wieder einsetzen zu können. Die Auswahl der Adresse erfordert, dass diese auf ein gültiges Befehlswort ausgerichtet ist und im Stack-Bereich liegt, um Kollisionen auf benachbarte Code-Segmente zu vermeiden. Zudem muss die Interpretation der Adresse und des zu schreibenden Sprungbefehls den aktiven *Adressierungsmodus*<sup>55</sup> berücksichtigen.

---

<sup>54</sup>Auch op code oder operation code, ist eine meist in hexadezimaler Schreibweise angegebene Zahl, die die Nummer eines Maschinenbefehls für einen bestimmten Prozessortyp angibt.

<sup>55</sup>Der Adressierungsmodus bezeichnet die Art und Weise, wie Operanden innerhalb des Befehlswortes adressiert werden; unterstützt werden u. a. Register-, symbolische, absolute und indirekte Adressierung. [11, S. 97, Kap. 4.4]

Die Implementierung gliedert sich in folgende Schritte:

1. **Adressvalidierung:** Prüfen, ob die Zieladresse auf vier Byte ausgerichtet ist und im Stack liegt.
2. **Kontext-Sicherung:** In einem kritischen Abschnitt werden PC, SP, SR sowie alle modifizierten Register in einem Puffer abgelegt. Hierbei ist zu beachten, dass die Breite des PC vom Adressierungsmodus abhängt, worauf entsprechend rücksicht genommen werden muss. [11, S. 97, Kap. 4.4]
3. **Ersetzen des Opcodes:** Der Originalen Opcode (vier Byte) wird mit dem vier-Byte-Sprungbefehl (zwei Byte für den Instruction-Code, zwei Byte für die Zieladresse der Handler-Routine) überschrieben. [11, S.161, Kap. 4.6.2.28]
4. **Ausführung des Breakpoint-Handlers:** Beim Eintreten des Breakpoint-Opcodes springt der PC in die Handler-Routine. Diese hindert das System an weiterer Ausführung, deaktiviert Interrupts und hält stattdessen das Observer-Modul für weitere Debug-Schritte aktiv.
5. **Kontext-Wiederherstellung:** Nach Abschluss der Debug-Aktion werden alle Register und der ursprüngliche Opcode wiederhergestellt, bevor der normale Programmablauf fortgesetzt wird.

Diese Vorgehensweise stellt die Rückkehr zum ursprünglichen Systemzustand – einschließlich des Originalen-Opcodes – sicher und müsste die Konsistenz des Programms garantieren.

Um die im vorigen Kapitel 3.5.2 skizzierten Konzepte robust umzusetzen, sind die im nächsten Abschnitt, technischen Details wie Instruktionslängen und Speicher-Alignment, zu betrachten.<sup>56</sup>

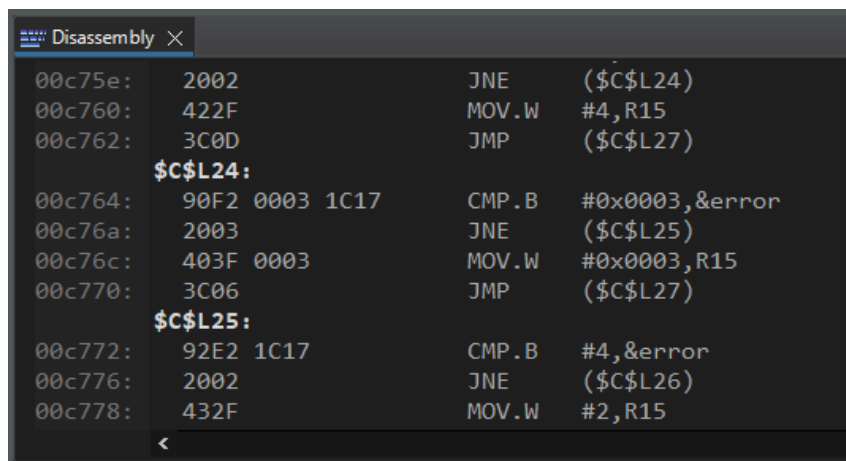
---

<sup>56</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

### 3.5.2.3. Instruktionslängen, Speicher-Alignment und Adress-Modi

Die Manipulation von Befehlen im Stack ist hoch kritisch, da das Hauptprogramm keine Kenntnis vom Observer-Modul hat und eine falsche Adressierung, unterschiedlichen Adressierungs-Modi oder unvollständige Opcode-substituierung zu undefiniertem Verhalten führen kann. Drei zentrale Aspekte sind dabei zu beachten:

- **Speicher-Alignment:** MSP430-Instruktionen sind grundsätzlich an geraden Speicheradressen ausgerichtet. Vor jedem Schreib- oder Lesezugriff muss daher sichergestellt werden, dass die Zieladresse eine gerade Adresse ist. Zugriffe auf ungerade Adressen können zu undefiniertem Verhalten führen, sofern keine Fehlerbehandlung erfolgt. Darüber hinaus sind Adressen, die in der Mitte eines Opcodes liegen, ungültig und dürfen nicht als Breakpoint adressiert werden.
- **Opcode-Längen:** Während einzelne Maschinenbefehle in der Regel zwei bis vier Bytes belegen, können komplexe Instruktionen – etwa `CMP.B` – bis zu sechs Bytes lang sein, wie in Abbildung 3.9 zu sehen. [11, S.165, Kap. 4.6.2.32] Diese Variabilität erschwert das gezielte Überschreiben von genau vier Byte, die für die Jump-Instruktion einschließlich Zieladresse benötigt werden. Unter diesen Randbedingungen besteht die Gefahr, dass entweder zu viele oder zu wenige Bytes überschrieben werden, was zu ungültigen oder unbeabsichtigten Instruktionen führen kann.



```
Disassembly x
00c75e: 2002      JNE      ($C$L24)
00c760: 422F      MOV.W   #4,R15
00c762: 3C0D      JMP      ($C$L27)
      $C$L24:
00c764: 90F2 0003 1C17  CMP.B   #0x0003,&error
00c76a: 2003      JNE      ($C$L25)
00c76c: 403F 0003      MOV.W   #0x0003,R15
00c770: 3C06      JMP      ($C$L27)
      $C$L25:
00c772: 92E2 1C17      CMP.B   #4,&error
00c776: 2002      JNE      ($C$L26)
00c778: 432F      MOV.W   #2,R15
```

Abb. 3.9.: Code Composer Disassembly Modus - Opcode längen

- **Adressierungs Modus:** Der Speicher – im Umfang von einem Megabyte – kann über sieben unterschiedliche Arten adressiert werden. Wahlweise kommen 16 oder 20-Bit Adressen zum Einsatz. Diese Variabilität der Adressierungsmodi und die damit einhergehenden unterschiedlichen Adressbreiten haben tiefgreifende Konsequenzen. [11, S. 97, Kap. 4.4]

Das Sichern und Wiederherstellen von Registern und des Originalen Opcodes reicht daher nicht aus, insbesondere angesichts variabler Instruktionslängen und komplexer Adressierungsmodi.

Die Größe der Opcodes hängt von folgenden Parametern ab. Die Grundlegende Länge einer Instruktion im MSP430 liegt bei zwei Byte. Darin ist die Operation, wie beispielsweise `MOV.W`, sowie die Register- und Adressierungsart kodiert. Unterschiedliche Adressierungen von Operanden über Adressierungsmodi wie „*Immediate*<sup>57</sup>“, „*Absolute*<sup>58</sup>“ oder „*Indirect*<sup>59</sup>“, haben zur Folge, dass die Adressen oder Konstanten zusätzlich als *Word*<sup>60</sup> (2 Byte) oder *extended Word*<sup>61</sup> (4 Byte) kodiert werden. [11, S. 97, Kap. 4.4]

Ein Beispiel hierzu, woraus sich der 6-Byte lange Opcode, zum Vergleichen des Speicherinhalts des Registers „&error“ mit dem Hexadezimalen Wert „0x0003“, aus Abbildung 3.9 zusammensetzt:

[11, S. 147, Kap. 4.6.2.14 & S. 112, Kap. 4.4.7 & S. 108, Kap. 4.4.4]

**CMP.B #0x0003, &error**

- **1 Byte** Opcode für `CMP.B`
- **1 Byte** Modus für Quell-Operand und Ziel-Operand (Immediate -> Absolute)
- **2 Byte** Immediate-Wert für Konstante „0x0003“
- **2 Byte** Absolute-Adresse für Label „&error“

<sup>57</sup>Operand ist direkt in der Instruktion enthalten – also ein fester Wert, nicht aus dem Speicher.

<sup>58</sup>Operand befindet sich an einer festen Speicheradresse.

<sup>59</sup>Operand ist nicht direkt gegeben, sondern steht an der Adresse, die ein Register enthält.

<sup>60</sup>Synonym für 2 Byte große Datenbreite.

<sup>61</sup>Synonym für 4 Byte große Datenbreite.

Diese Analyse verdeutlicht die Komplexität der Manipulation und Wiederherstellung von Instruktionen im Stack. Eine Kopie des Stacks schafft dabei eine sichere Umgebung, in welcher der bestehende Opcode manipuliert wird. Hierbei muss die Kopie die unterschiedlichen Adressformate und die potenziell im Stack gespeicherten, modusabhängigen Zeiger korrekt abbilden oder transformieren. Dies garantiert ein sicheres zurückkehren in die Hauptroutinen, sowie eine robuste Ausführung der Funktion zum Verarbeiten der ggf. mehreren Breakpoints.

Dies erschwert die Umsetzung und erhöht die Komplexität der Routine erheblich, wodurch Timing und Konsistenz gefährdet werden. Im anschließenden Fazit werden die gewonnenen Erkenntnisse bewertet, offene Fragestellungen skizziert und ein Ausblick auf weiterführende Arbeiten gegeben.



## 4. Fazit und kritische Bewertung

Im Rahmen dieser Arbeit wurde eine interruptgesteuerte Benutzerschnittstelle für den Mikrocontroller MSP430FR5729 konzipiert und implementiert. Zentrale Komponente ist das sogenannte Observer-Modul, welches sowohl den Empfang als auch die statusbasierte Verarbeitung externer Steuerbefehle übernimmt. Damit bildet es eine funktionale Schnittstelle zwischen dem Mikrocontroller, seinem Hauptprogramm und der Benutzerinteraktion.

### 4.1. Das Ergebnis

Die Entwicklung wesentlicher Bausteine, insbesondere die der Timer-Interrupt-Service-Routine, UART-Kommunikationsschnittstelle sowie der statusorientierten Abarbeitung eingehender Befehle, legt eine flexible und erweiterbare Basis für zukünftige Funktionsimplementierungen. Die exemplarisch realisierten Lese- und Schreibfunktionen demonstrieren die praktische Anwendbarkeit dieser Architektur.

Ein bedeutendes Ergebnis stellt die Realisierung des Observer-Moduls für Debugging-Zwecke dar, welches unabhängig von der üblichen Entwicklungsumgebung (Code Composer Studio und MSP-FET) agiert. Dies erlaubt es dem Benutzer, unter Realbetriebsbedingungen, tiefgreifende Analysen durch das Auslesen und Beschreiben von Speicherzellen während der aktiven Laufzeit. Gerade bei der Entwicklung auf Mikrocontrollern – wie dem MSP430 – sind Stichproben unter Echtzeitbedingungen von hohem Wert, da kontrolliert und nicht blockierend auf Speicherzellen zugegriffen werden kann.

Im Anschluss an die Implementierung wurde ein besonderes Augenmerk auf die Analyse der Systemreaktivität gelegt. Zu diesem Zweck wurden Laufzeitmessungen durchgeführt, welche aufzeigen, inwieweit die entwickelten Komponenten den Anforderungen an Echtzeitfähigkeit genügen. Das folgende Kapitel beschreibt detailliert die zugrunde liegende Methodik und präsentiert die ermittelten Messergebnisse.<sup>62</sup>

---

<sup>62</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

#### 4.1.1. Untersuchung der Laufzeit unter Realbedingungen

Zur Untermauerung der in Kapitel 3.3.1 genannten Vorteile eines statusorientierten Zustandsautomaten hinsichtlich der Systemreaktivität sind Laufzeituntersuchungen unerlässlich. Diese Messungen ermöglichen eine fundierte Bewertung der Ausführungszeiten hinsichtlich genannter Vor- und Nachteile für andere Softwarekomponenten und Systemzustände.

Die Erfassung und Auswertung der Laufzeiten erfolgte mittels eines Digitalen Oszilloskops (Modell: PicoScope 3406D MSO) in Verbindung mit der zugehörigen Messsoftware PicoScope 7 T&M. Für die Aufzeichnung der Signale wurde der digitale Eingang D1 des Oszilloskops genutzt. Eine einheitliche Abtastrate von 500 *MS/s* (Mega-Samples pro Sekunde) wurde für alle Messungen verwendet, um eine hohe zeitliche Auflösung sicherzustellen.

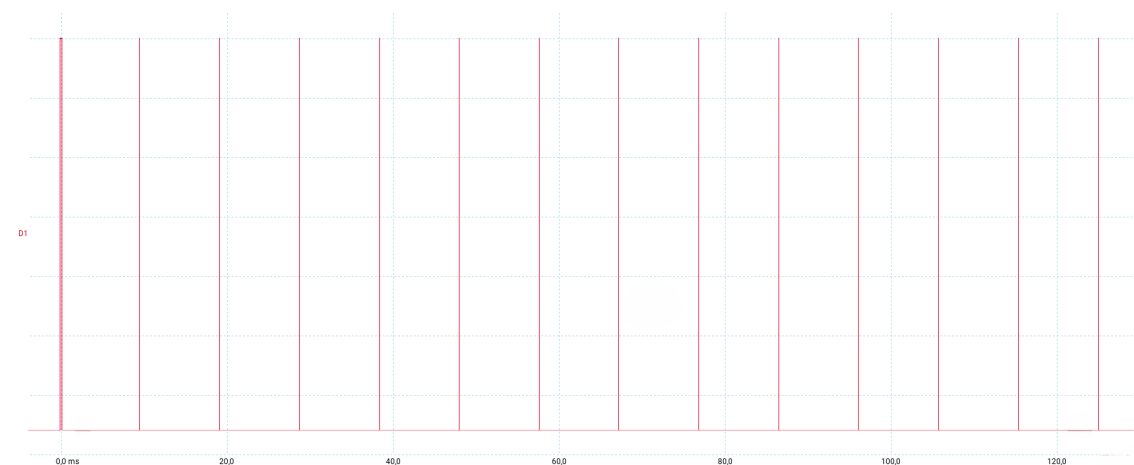
Zur Signalisierung der Messintervalle wurden GPIO-Pins von Port 3 des zugrundeliegenden Mikrocontrollers verwendet. Die exakte Zeitmessung einzelner Codeabschnitte erfolgte durch das Setzen eines spezifischen GPIO-Pins (konkret P3.4, adressiert über BIT4 des P3OUT-Registers) zu Beginn des zu messenden Abschnitts (`SETBIT(P3OUT, BIT4);`) und dessen unmittelbares Zurücksetzen (`CLRBIT(P3OUT, BIT4);`) an dessen Ende. Diese Implementierung ist in den auskommentierten Zeilen von Listing 3.8 und Listing 3.9 sowie in den Codeausschnitten der Zustände aus Kapitel 3.2 ersichtlich. Dieses Verfahren ermöglicht eine präzise Erfassung der Ausführungsdauer sowohl einzelner Verarbeitungszyklen als auch kompletter Lese- oder Schreiboperationen.

Die nachfolgende Tabelle 4.1 fasst die ermittelten Laufzeiten für repräsentative Ereignisse und Funktionen innerhalb des Observer-Moduls zusammen.

Ereignis	Funktion	Laufzeit
Leerlauf	Timer-ISR	~3,9 $\mu$ s
Time-Out Timer	state_0	~3,4 $\mu$ s
Befehlsinterpretation	state_1	~70 - 140 ms
Resetzustand	state_2	~6,3 $\mu$ s
Leseoperation für 13 Byte	read_mem	~130 ms
Leseoperation Parameterentnahme	read_mem	~240 $\mu$ s
Leseoperation pro Byte	read_mem	~10 $\mu$ s (+ ~10 ms ISR Zyklus)
Schreiboperation für „test“	write_mem	~40 ms
Schreiboperation Parameterentnahme	write_mem	~220 $\mu$ s
Schreiboperation pro Byte	write_mem	~10 $\mu$ s (+ ~10 ms ISR Zyklus)

**Tab. 4.1.:** Laufzeitmessungen – Observer-Modul State-Machine

Ein wesentliches Ergebnis der statusorientierten, zeichenweisen Verarbeitung ist die Freigabe von Rechenzeit zwischen den einzelnen Verarbeitungszyklen für andere Systemaufgaben. Abbildung 4.1 visualisiert diesen Sachverhalt exemplarisch anhand einer Leseoperation.



**Abb. 4.1.:** Oszillogramm einer vollständigen Speicherleseoperation (rdm 0x1C00 13)

Jeder in der Abbildung dargestellte Impuls repräsentiert die Verarbeitungszeit für ein einzelnes Zeichen. Die dargestellte Operation (`rdm 0x1C00 13`) weist insgesamt 14 solcher Impulse auf. Der initiale Impuls zeigt die längste Dauer, welche aus der einmaligen Initialisierungsphase pro Leseoperation resultiert: Hierbei werden die erforderlichen Parameter (Speicheradresse, Anzahl der Bytes) aus dem `uart_buffer` extrahiert. Dieser Vorgang involviert rechenzeitintensive String-Operationen wie `strtok`, `strlen`, `strtol` und `atoi`. Die nachfolgenden Zyklen, welche die eigentliche zeichenweise Leseoperation des Zeichens umfassen, weisen signifikant kürzere Ausführungszeiten von etwa 10 µs auf. In den Intervallen – von ca. 10 ms – zwischen diesen aktiven Verarbeitungsphasen (bedingt durch die Periode der Timer-ISR) steht der Prozessor für andere Aufgaben – wie beispielsweise der Ausgabe des gelesenen Zeichens – zur Verfügung. Der letzte Verarbeitungszyklus (in Abbildung 4.1 der 14.) dient der Finalisierung der Leseoperation. Er beinhaltet die erfolgreiche Prüfung auf die angeforderte Anzahl an gelesenen Zeichen (Abgleich von `mem_addr_idx` mit `blocks` in der `read_mem`-Funktion aus Listing 3.8), und leitet bei Erfolg den Übergang in den Rücksetzzustand (`state_2`) durch Aktualisierung des `state_ptr` ein.

Auch die in Kapitel 3 beschriebenen Vorteile der Integration softwarebasierter Breakpoints – insbesondere für kritische Operationen wie das Lesen und Schreiben auf Speicherzellen – konnten im Verlauf der Entwicklung bestätigt werden. Die in Kapitel 3.5 begonnene Konzeptausarbeitung liefert eine fundierte Grundlage für die folgende Bewertung dieser Mechanismen.<sup>63</sup>

---

<sup>63</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

### 4.1.2. Fazit zu Software Breakpoints

Die Realisierung von Software-Breakpoints auf dem Low-Power-Mikrocontroller MSP430FR5729 erfordert ein tiefgehendes Verständnis der Prozessor-Architektur, der Instruktionsformate, vielfältiger Adressierungs-Modi und ihrer Auswirkungen auf die Speicherverwaltung sowie der nebenläufigen Abläufe im System. Die Analyse der Problemstellung hat ergeben, dass unter anderem kritische Bereiche atomar bearbeitet, Register und Stack-Zustände zuverlässig gesichert und Intrinsics korrekt eingesetzt werden müssen. Zudem sind umfangreiche Funktionen zur Überwachung und Protokollierung des Systemzustands zu implementieren.

Die erwartete Komplexität inklusive der Anforderungen an Robustheit, Echtzeitfähigkeit und möglichst geringem Eingriff in den Betrieb überschreitet den Rahmen einer üblichen Bachelorarbeit. Eine vollständige, ausgereifte Implementierung wäre mit dem Umfang einer Masterarbeit oder vergleichbarer Forschungsarbeiten möglich. Dennoch bildet dieses Thema eine exzellente Grundlage für weiterführende Arbeiten in den Bereichen eingebetteter Systeme und Debugging-Technologien, insbesondere in Bezug auf Architekturen mit flexiblen, aber komplexen Speicheradressierungsmechanismen.

## 4.2. Die Bewertung

Die zeitkritische Synchronisation zwischen Hauptprogramm und Observer-Modul stellte eine zentrale Herausforderung dar – insbesondere im Hinblick auf den gewählten, nicht-blockierenden Ansatz der statusorientierten Verarbeitung. Die sequentielle, zeichenweise Abarbeitung von Lese- und Schreiboperationen über mehrere Verarbeitungsschritte hinweg erhöhte zwar die Komplexität der Implementierung, führte jedoch zu einer äußerst performanten Lösung hinsichtlich der Systemreaktivität. Dies ist insbesondere für ein als „unsichtbares“ Plug-in-Modul konzipiertes System essenziell.

Die vollständige Implementierung softwaregesteuerter Breakpoints erweist sich hingegen als äußerst anspruchsvoll. Ein vielversprechender Ansatz wird, gemäß dem in Kapitel 3.5.2.2 skizzierten Konzept, entwickelt, (Abbildung 4.2) jedoch gelang keine vollständig funktionale, robuste und konsistente Einbindung zur Laufzeit. Der entsprechende Quellcode befindet sich, wenn auch auskommentiert, im Observer-Modul.

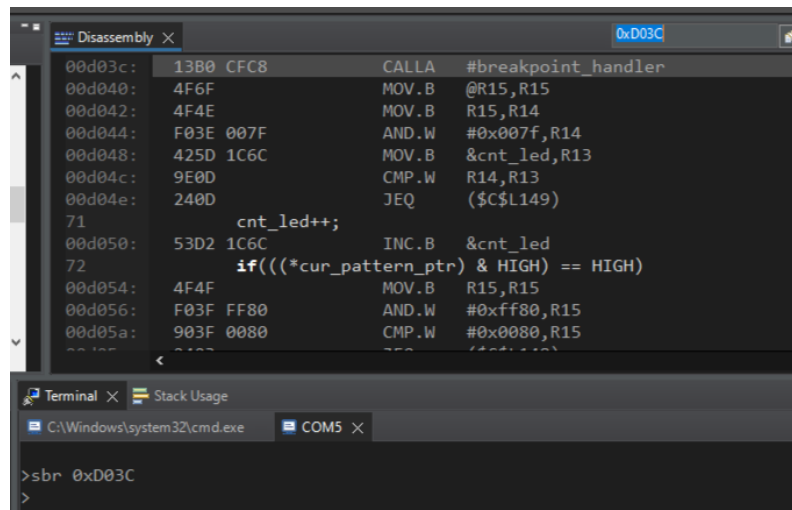


Abb. 4.2.: Setzen eines Software-Breakpoints an einer frei ausgewählten Speicheradresse im Stack – Befehl: `sbr 0xD03C`

Nichtsdestotrotz konnte durch die umfassend durchgeführte Laufzeitanalyse die Echtzeitfähigkeit des Observer-Moduls bestätigt werden. Diese wird allerdings mit einem erhöhten Speicherverbrauch erkauft, bedingt durch den Einsatz nicht vollständig optimierten Codes und speicherintensiver Funktionen der Standard-String-Bibliothek.

Ein wesentliches Manko stellt die aktuelle Beeinträchtigung der UART-Schnittstelle dar. Aufgrund der nicht implementierten automatischen Baudratenerkennung – wie in Kapitel 2.2.3.3 kurz beleuchtet wurde – kann das Hauptprogramm diese Schnittstelle nicht parallel nutzen. Dies steht dem Anspruch entgegen, das Observer-Modul vollständig vom restlichen System zu entkoppeln, da ansonsten ein zentrales Modul für Kommunikationsmanagement zur Konfliktvermeidung erforderlich wäre.<sup>64</sup>

<sup>64</sup>Die sprachliche Überarbeitung dieses Kapitels erfolgt unter Zuhilfenahme von KI-Sprachmodellen. Inhaltliche Aussagen und Schlussfolgerungen stammen ausschließlich vom Autor.

### 4.3. Ein Ausblick

Der modulare Aufbau des Systems bietet, kombiniert mit der flexiblen Konfiguration der Timer- und UART-Interrupts, eine solide Grundlage für die Integration weiterführender Funktionalitäten. Insbesondere die strukturierte Umsetzung und umfangreiche Analyse der Software-Breakpoint-Mechanismen liefern wertvolle Impulse für zukünftige Arbeiten im Bereich experimenteller Debugging-Techniken.

Ein weiterer Entwicklungsschritt könnte die Implementierung eines weiteren Moduls zur automatischen Baudratenerkennung darstellen. Diese würde es ermöglichen, die UART-Schnittstelle sowohl für interne als auch externe Kommunikationsprozesse effizient zu nutzen, ohne die Systemtrennung aufzugeben.

Langfristig könnte die entwickelte Architektur auch in komplexere, multitaskingfähige Systeme überführt werden. Durch den nicht-blockierenden und zustandsbasierten Aufbau ist das Observer-Modul prädestiniert für eine Integration in reaktive, ereignisgesteuerte Steuerungsplattformen zur Ergänzung bereits vorhandener Debugging-Mechanismen.

## Literatur

- [1] Texas Instruments. *Code Composer User's Guide*. Document Number: SPRU296. Texas Instruments Incorporated. 655303 Dallas, Texas, 1999, S. 224. URL: <https://www.ti.com/lit/ug/spru296/spru296.pdf>.
- [2] Miquel van Smoorenburg. *Linux Serial Console*. 2000. URL: <https://www.kernel.org/doc/html/latest/admin-guide/serial-console.html> (besucht am 12.06.2025).
- [3] DS Yadav. *Microcontroller: features and applications*. New Age International, 2004.
- [4] John H. Davies. *MSP430 Microcontroller Basics*. 1. Aufl. Oxford: Newnes, 2008. ISBN: 978-0-7506-8276-3.
- [5] Heinrich Müller u. a. „Rechnerarchitektur und Maschinensprache“. In: *Vorkurs Informatik: Der Einstieg ins Informatikstudium* (2012), S. 259–266.
- [6] Universal Asynchronous Receiver-Transmitter. „UART“. In: (2016). URL: <https://www.arxterra.com/wp-content/uploads/2018/06/UART.pdf>.
- [7] Elektronik-Kompodium. *Logik-Pegel*. 2017. URL: <https://www.elektronik-kompodium.de/sites/dig/0205171.htm> (besucht am 18.06.2025).
- [8] Texas Instruments. *MSP430FR5729 Mixed-Signal Microcontroller*. Revision C. Document Number: SLASE35. Texas Instruments Incorporated. 655303 Dallas, Texas, 2017, S. 119. URL: <https://www.ti.com/lit/ds/symlink/msp430fr5729.pdf>.
- [9] William G. Wong. *What's the Difference Between NRZ, NRZI, and Manchester Encoding?* 2017. URL: <https://www.electronicdesign.com/technologies/communications/article/21802271/electronic-design-whats-the-difference-between-nrz-nrzi-and-manchester-encoding> (besucht am 18.06.2025).
- [10] Texas Instruments. *MSP430 Optimizing C/C++ Compiler v18.1.0.LTS*. Revision R. Document Number: SLAU132, Rev. R. Texas Instruments Incorporated. 655303 Dallas, Texas, 2018, S. 181. URL: <https://www.ti.com/lit/ug/slau132r/slau132r.pdf>.



- 
- [11] Texas Instruments. *MSP430FR57xx Family User's Guide*. Revision D. Document Number: SLAU272, Rev. D. Texas Instruments Incorporated. 655303 Dallas, Texas, 2018, S. 576. URL: <https://www.ti.com/lit/ug/slau272d/slau272d.pdf>.
- [12] Simon Oelgeschläger. „SRAM und DRAM-Eine Übersicht“. In: (2019).
- [13] Texas Instruments. *MSP Flasher user's guide*. Revision E. Document Number: SLAU654. Texas Instruments Incorporated. 655303 Dallas, Texas, 2019, S. 18. URL: <https://www.ti.com/lit/ug/slau654e/slau654e.pdf>.
- [14] Roshni Y. *RISC Processor: Architecture, Advantages and Disadvantages*. 2020. URL: <https://electronicsdesk.com/risc-processor.html> (besucht am 18.06.2025).
- [15] Texas Instruments. *Code Composer Studio™ IDE v10.x for MSP430™ MCUs*. Revision AS. Document Number: SLAU157. Texas Instruments Incorporated. 655303 Dallas, Texas, 2020, S. 66. URL: <https://www.ti.com/lit/ug/slau157as/slau157as.pdf>.
- [16] mikecoats.com. *Connecting to Serial Ports with Windows Terminal*. 2024. URL: <https://mikecoats.com/serial-windows-terminal/> (besucht am 12.06.2025).
- [17] Robert Sheldon. *MHz (Megahertz)*. 2024. URL: <https://www.computerweekly.com/de/definition/MHz-Megahertz> (besucht am 18.06.2025).
- [18] ComputerNetworkingNotes. *Simplex, Half-duplex, and, Full-duplex Explained*. 2025. URL: <https://www.computernetworkingnotes.com/networking-tutorials/simplex-half-duplex-and-full-duplex-explained.html> (besucht am 18.06.2025).
- [19] Duden. *Plug and Play*. 2025. URL: <https://www.duden.de/node/112521/revision/1870956> (besucht am 18.06.2025).
- [20] Martin Falch. *LIN Bus Protocol: An Introduction*. CSS Electronics, 2025. URL: <https://www.csselectronics.com/pages/lin-bus-protocol-intro-basics> (besucht am 12.06.2025).
- [21] Riverdi Sp. z o.o. *Baudrate verstehen: Ein umfassender Leitfaden*. 2025. URL: <https://riverdi.com/de/blog/ baudrate-verstehen-ein-umfassender-leitfaden> (besucht am 12.06.2025).

- 
- [22] Texas Instruments. *Code Composer Studio™ User's Guide*. Version 20.1.1. Texas Instruments Incorporated. 655303 Dallas, Texas, 2025. URL: [https://software-dl.ti.com/ccs/esd/documents/users\\_guide/index.html](https://software-dl.ti.com/ccs/esd/documents/users_guide/index.html).
- [23] Texas Instruments. *MSP-FET - MSP MCU Programmer and Debugger*. 2025. URL: <https://www.ti.com/tool/MSP-FET> (besucht am 22.05.2025).

# Abbildungsverzeichnis

2.1. Operating Modes [11, S. 37, Kap. 1.4, Tab. 1-2] . . . . .	3
2.2. Block Diagramm MSP430FR5729 Mikrocontroller [8, S. 2, Kap. 1.4] .	5
2.3. Up Mode [11, S. 359, Abb. 12.2] . . . . .	7
2.4. Continuous Mode [11, S. 360, Abb. 12.4] . . . . .	8
2.5. Up/Down Mode [11, S. 361, Abb. 12.7] . . . . .	8
2.6. Capture Mode Einsatzbeispiele [4, S. 301, Abb. 8.7] . . . . .	9
2.7. Ausgabeeinheit im Up/Down-Modus [11, S. 340, Abb. 11-9] . . . . .	12
2.8. Timer B Block & Capture/Compare Channel 1 [4, S. 355, Kap. 8.16]	19
2.9. UART Übertragung der Werte 0x55 und 0xFF [4, S. 576, Abb. 10.18]	24
2.10. UART Übertragung der Werte 0x55 und 0xFF [4, S. 577, Abb. 10.19]	26
2.11. Automatische Baudratenerkennung - Break/Sync Sequenz [11, S. 481, Abb. 18-5] . . . . .	28
2.12. Automatische Baudratenerkennung - Sync Feld [11, S. 481, Abb. 18-6]	28
2.13. eUSCI Typ A – UART-Modus [11, S. 477, Kap. 18.2] . . . . .	32
3.1. UML-Diagram – Observer-Modul . . . . .	34
3.2. Zustandsautomat – Observer-Modul . . . . .	35
3.3. Aktivitätsdiagramm – Timer ISR . . . . .	39
3.4. Aktivitätsdiagramm – Timer ISR . . . . .	41
3.5. Aktivitätsdiagramm – Timer ISR . . . . .	42
3.6. Memory map des FG4618 [4, S. 603, Fig. 11.1] . . . . .	51
3.7. Flash Emulation Tool Programmer and Debugger [23] . . . . .	61
3.8. Code Composer Studio - Breakpoint Übersicht . . . . .	62
3.9. Code Composer Disassembly Modus - Opcode längen . . . . .	66
4.1. Oszillogramm einer vollständigen Speicherleseoperation ( <code>rdm 0x1C00</code> <code>13</code> ) . . . . .	71
4.2. Setzen eines Software-Breakpoints an einer frei ausgewählten Spei- cheradresse im Stack – Befehl: <code>sbr 0xD03C</code> . . . . .	74

## Tabellenverzeichnis

2.1. Registerbeschreibung – Capture-/Compare Register Timer B [11, S. 375, Tab. 12-8] . . . . .	13
2.2. Registerbeschreibung – Control Register Timer B [11, S. 372, Tab. 12-6]	16
2.3. Funktionsvergleich der eUSCI-Module des MSP430FR5729 [11, 4, Kap. 18, 19, 20, S. 493, Kap. 10] . . . . .	21
2.4. Vergleich der synchronen seriellen Protokolle SPI und I <sup>2</sup> C [4, S. 497, Kap. 10.2, S. 534, Kap. 10.7] . . . . .	22
2.5. UART-Fehlerbedingungen und zugehörige Status-Flags des MSP430FR5729 [11, S. 483, Tab. 18-1] . . . . .	27
2.6. Technische Merkmale der UART-Schnittstelle des MSP430FR5729 [11, S. 476, kap. 18.2] . . . . .	27
4.1. Laufzeitmessungen – Observer-Modul State-Machine . . . . .	71

## Verzeichnis der Listings

3.1. Funktionszeiger für die Zustandsmaschine . . . . .	43
3.2. Implementierung – state_dummy . . . . .	43
3.3. Implementierung – state_0 . . . . .	44
3.4. Implementierung – state_1 . . . . .	44
3.5. Implementierung – state_2 . . . . .	45
3.6. Dictionary für Befehls-Funktionszeiger verknüpfung . . . . .	45
3.7. Initiierungs-Funktion für UART Datenübertragung . . . . .	46
3.8. Implementierungsausschnitt der <code>read_mem</code> -Funktion im Observer-Modul	50
3.9. Implementierungsausschnitt der <code>write_mem</code> -Funktion im Observer- Modul . . . . .	52
3.10. Zeichenprüfung auf Alphanumerik mittels eigen Makros . . . . .	53
3.11. Funktion zur Fehlerpriorisierung und Speicherung sowie Setzen eines globalen <code>Error</code> -Flags . . . . .	55
3.12. Definition der Fehler-Makros in der Header-Datei <code>Observer.h</code> zur Festlegung der Fehlerprioritäten. . . . .	56
3.13. Zusammensetzung und Initialisierung der Fehlernachrichtausgabe . .	57

## A. Anhang

### A.1. Verwendete Hilfsmittel

#### A.1.1. Erklärung zur Nutzung von KI-Sprachmodellen zur sprachlichen Überarbeitung

Im Rahmen dieser Bachelorarbeit wurden **Large Language Models**, namentlich ChatGPT und GeminiAI, zur kritischen Bewertung und konstruktiven Überarbeitung der sprachlichen Ausdrucksweise eingesetzt. Dies umfasst stilistische und Grammatikalische Optimierung in folgenden Bereichen:

- Verbesserung der Lesbarkeit und sprachlichen Klarheit,
- Vereinheitlichung des wissenschaftlichen Ausdrucks,
- Korrekturvorschläge von Grammatik-, Rechtschreibung- und Zeichensetzung.

Alle Kapitel, die überarbeitet werden, sind am Ende mit einem Hinweis in einer Fußnote gekennzeichnet.

Die inhaltliche Verantwortung für alle Aussagen, Argumentationen und wissenschaftlichen Schlussfolgerungen liegt vollständig bei dem Autor. Die genannten KI-Modelle werden **nicht** zur Generierung fachlicher Inhalte, zur Datenanalyse oder zur Strukturierung argumentativer Abschnitte verwendet.

Die Verwendung dieser Werkzeuge erfolgt unter Beachtung geltender ethischer Richtlinien für wissenschaftliche Arbeiten sowie der Anforderung an Eigenständigkeit und Transparenz.

### A.1.2. Erklärung zur Erstellung von Diagrammen

Zur Erstellung von Diagrammen wird das Tool *PlantUML* (verfügbar unter <https://www.plantuml.com>) verwendet.

Mit Hilfe des Tools werden verschiedene UML-Diagramme wie Aktivitätsdiagramme, Klassendiagramme und Zustandsdiagramme (State-Machine-Diagramme) modelliert. PlantUML ermöglicht die textuelle Beschreibung von Diagrammen, welche anschließend automatisch als grafische Darstellung generiert werden. Dies erleichtert sowohl die Versionierung als auch die Nachvollziehbarkeit der Diagrammerstellung im Entwicklungsprozess.