Entwicklung eines JavaScript-Demonstrators für die Compare-Einheit des MSP430

Studienarbeit

vorgelegt im

Studiengang Elektrotechnik

Duale Hochschule Baden-Württemberg
Mannheim

von

Name, Vorname	van den Boom, Tim
Abgabedatum	25.12.2025
Bearbeitungszeitraum	29.09.2025 — 02.01.2026
Ausbildungsfirma	DB InfraGO AG, Karlsruhe
Matrikelnr., Kurs	6250261, TEL23AEO
Betreuer der Hochschule	Prof. Dr. Heintz, Rüdiger

Unterschrift Betreuer

1 Kurzfassung

Die vorliegende Arbeit dokumentiert die Entwicklung eines HTML/JavaScript-Demonstrators für die Compare-Einheit des MSP430 von Texas Instruments. Er dient Studierenden als interaktive Möglichkeit, Timer-Grenzen und bis zu drei Ausgangskanäle zu konfigurieren und das Verhalten der Timer-Einheit praxisnah zu beobachten.

Der Demonstrator stellt die in der offiziellen MSP430-Dokumentation beschriebenen Grafiken als Simulation dar und bietet didaktische Hinweise, um den Einstieg in die hardwarenahe Programmierung zu erleichtern und auf fehlerhafte Eingaben hinzuweisen.

Für die korrekte Simulation wurden Messungen mit einem Oszilloskop durchgeführt, um widersprüchliche Angaben in der Dokumentation zu überprüfen und einen realistischen Demonstrator zu entwickeln.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Codeblöcke

Tabellenverzeichnis

2 Einleitung

Zunächst beginnt diese Arbeit mit einer Einleitung in das Projekt der Studienarbeit. Dazu zählt zunächst eine kurze Vorstellung des Mikrocontroller-Labors allgemein sowie die für dieses Projekt zu betrachtenden, bereits bestehenden Folien. Zuletzt wird aus dieser Einordnung ein Ziel an diese Arbeit gestellt.

2.1 Das Mikrocontroller-Labor

Diese Studienarbeit entsteht im Rahmen des Mikrocontroller-Labors von Prof. Dr. Heintz an der DHBW Mannheim. Für den Studiengang Elektrotechnik ist es für das dritte Semester vorgesehen und lässt sich dem Modul *High Level Synthesis* zuordnen. Als gesamt aufzuwendende Zeit sind 44 Stunden vorgesehen, davon 14 in Präsenz. Die Präsenztermine werden vorrangig zur Klärung von Fragen und der Durchführung der notwendigen Testate verwendet. Insgesamt sind fünf Testate erfolgreich zu absolvieren. Der den Studenten zur Verfügung gestellte Foliensatz besteht einerseits aus erklärenden Inhalten, andererseits werden dort auch konkrete Aufgabenstellungen formuliert, welche die Basis für den einzureichenden Programmcode bilden.

Da der Großteil der vorgesehenen Arbeitszeit dem Selbststudium zugeordnet ist, wird kein realer Mikrocontroller verwendet. Stattdessen erhalten die Studierenden eine virtuelle Java-Simulation, die orts- und hardwareunabhängig ausgeführt werden kann. Dennoch erfolgt die Simulation hardwarenah und mit realen Registern und Schnittstellen. Für die Versuche wird die Experimentierplatine `MSP430F5529LP` genutzt, welche einen `MSP430F5529` von Texas Instruments integriert und vielfältige Peripheriemodule wie Timer, Analog/digital-Wandler und serielle Schnittstellen bereitstellt.

2.2 Stand der Foliensätze

Zu Beginn jeder Aufgabe erhalten die Studierenden Informationen über die zu erfüllenden Aufgaben. Die Basis für diese Arbeit bietet ein interaktiver HTML-Foliensatz, den die Studierenden im Browser aufrufen können. Entsprechend ergibt sich die Möglichkeit der Einbettung von Simulationen und Demonstratoren. Beispielhaft für diese Einbettungen sind etwa der Demonstrator für bitweise Maskierung oder Interrupt-Routinen zu nennen.

Der in dieser Arbeit zu entwerfende Demonstrator ist in Versuch zwei zu verorten. Konkret sollen sich die Studierenden mit dem Timer des MSP430 vertraut machen sowie zuletzt ein PWM-Signal für einen Piezo-Hochtöner erzeugen. In den Vorbetrachtungen soll dazu der Timer konfiguriert werden, der das Signal über die Compare-Einheit der CCU entwickelt.

Für den erfolgreichen Abschluss der Aufgabe ist ein genaues Verständnis der Funktionen der Capture-Compare-Einheit entscheidend, damit die Timer-Grenzen korrekt eingestellt und sowohl eine LED, ein Display und später der Piezo-Kristall sinnvoll angesteuert werden können. Der aktuelle Foliensatz präsentiert dazu in einer Grafik den logischen Innenaufbau des Timers, als auch die Schaltung hinter der Signalerzeugung der CCU. Zu beiden Schaubildern existiert bereits eine etwa 35 Minuten lange Erklärung in Videoform.

2.3 Aufgabenstellung

Zusätzlich zu diesen Erklärungen sollen die Studierenden jedoch auch mit eigens festgelegten Werten anschaulich experimentieren können. Optimal geeignet sind hierfür entsprechende Zeit-Werte-Diagramme, bei denen die Studierenden die Auswirkungen der verschiedenen Timer- und Modi-Konstellationen anschaulich simulieren können. Ziel dieser Arbeit ist eine Vollbild-Simulation auf einer der dem Versuch zwei zugehörigen Folien. Es bietet sich an, diese Folie entsprechend den Erklärungen zum Timer und zur CCU zuzuordnen.

Weil ohne externe Interrupts keine virtuelle Capture-Unit simuliert werden kann, beschränkt sich das Ziel dieser Arbeit grundsätzlich in das Erzeugen von PWM-Signalen mit den vom Timer A zur Verfügung gestellten Mitteln. Weiterhin soll zwischen zwei Taktgebern unterschiedlicher Frequenz gewählt werden können, und das Einstellen der Sollwert-Register `TAxCCR0` – `TAxCCR2` soll ermöglicht werden. Es ergeben sich die simulierten Signale `OUT0` - `OUT2`. Eine Übersicht über die technischen Merkmale dieses Timers bieten die folgenden Kapitel.

Die spätere Einbettung des hier entwickelten Demonstrators erfolgt über ein HTML-Iframe, sodass bereits in der Vorbetrachtung auf die korrekte Geometrie zu achten ist. Weil der Demonstrator folienfüllend sein soll, ist das Querformat zu verwenden.

2.4 Aufbau der Arbeit

In den theoretischen Grundlagen werden zunächst der Mikrocontroller MSP430 sowie dessen Capture-Compare-Einheit erläutert. Anschließend wird auf die im Labor verwendete Variante *MSP430F5529* sowie auf die Produktfamilien *MSP430x5xx* und *MSP430x6xx* eingegangen. Der Theorieteil schließt mit einem kurzen Überblick über grundlegende Konzepte der Webprogrammierung.

Der praktische Teil der Arbeit befasst sich mit der Konzeption und Umsetzung. Zunächst werden grundlegende Vorbetrachtungen hinsichtlich des Layouts, der Variablenstruktur sowie der Verarbeitung von Nutzerdaten dargestellt. Darauf aufbauend werden die Simulation und Weiterverarbeitung der Timer-Signale beschrieben. Abschließend wird die Generierung eines maschinenlesbaren Ausgangsformats für verschiedene Betriebsmodi des MSP430 erläutert. Ergänzend wird die Herstellerdokumentation im Hinblick auf ihre inhaltliche Konsistenz analysiert.

3 Theoretische Grundlagen

Dieses Kapitel vermittelt die für das Verständnis der Arbeit notwendigen theoretischen Grundlagen. Zunächst wird ein Überblick über die Mikrocontroller-Produktfamilie MSP430 gegeben, wobei allgemeine technische Eigenschaften sowie Unterschiede zwischen verschiedenen Derivaten betrachtet werden. Darauf aufbauend werden die Funktionsweise und Einsatzmöglichkeiten der Capture-Compare-Einheit erläutert, mit besonderem Fokus auf die in dieser Arbeit verwendeten Timer-Module. Ergänzend werden die spezifischen Eigenschaften des im Labor eingesetzten Mikrocontrollers *MSP430F5529* dargestellt. Abschließend bietet das Kapitel eine Einführung in grundlegende Konzepte der Webprogrammierung, die für die Umsetzung der entwickelten Lösung relevant sind.

3.1 Die MSP430-Produktfamilie

In diesem Abschnitt wird die MSP430-Produktfamilie überblicksartig vorgestellt. Dabei werden zunächst allgemeine technische Merkmale betrachtet, bevor anschließend auf spezifische Eigenschaften einzelner Derivate eingegangen wird.

3.1.1 | Allgemeine technische Daten

Die Controllerfamilie MSP430 basiert grundlegend auf einer Von-Neumann-Architektur. Die Namensgebung *MSP* (Mixed Signal Processor) verweist dabei auf die interne Struktur der Bausteine, welche für die Verarbeitung sowohl analoger als auch digitaler Signale optimiert ist. Ein wesentliches Merkmal dieser 16-Bit-Mikrocontroller ist ihr niedriger Stromverbrauch. Diese Eigenschaft prädestiniert die MSP430-Familie für mobile Einsatzgebiete, in denen ein möglichst geringer Energiebedarf von entscheidender Bedeutung ist. [1, S. 41]

Zentrale Einheit des Mikrocontrollers ist die CPU. Während frühere Modelle der *3xx*-Serie sowie die in diesem Kontext betrachteten Standard-Varianten über eine 16-Bit-CPU verfügen, wurde die Architektur später um die sogenannte *CPUX* erweitert, die eine 20-Bit-Verarbeitung ermöglicht und dabei volle Rückwärtskompatibilität zur ursprünglichen 16-Bit-Architektur wahrt. [2, S. 95] Die CPU ist als *Reduced Instruction Set Computer* (RISC) konzipiert. Der Befehlssatz ist hierbei stark reduziert und umfasst lediglich 27 Grundbefehle sowie 24 weitere emulierte Befehle. [1, S. 41] [2, S. 95]

Die Architektur verfügt über insgesamt 16 Register mit einer Breite von jeweils 16 Bit (R0 bis R15). Von diesen 16 Registern sind vier für dedizierte Aufgaben reserviert (*Special Purpose Registers*):

- `R0 (PC)` : Der Befehlszähler (Program Counter).
- `R1 (SP)` : Der Stapelzeiger (Stack Pointer).
- `R2 (SR/CG1)` : Das Statusregister, welches in einer Doppelfunktion auch als Konstantengenerator dient.
- `R3 (CG2)` : Ein reiner Konstantengenerator für Instruktionskonstanten, der nicht zur Datenspeicherung verwendet wird.

Die verbleibenden Register R4 bis R15 stehen als Mehrzweckregister (*General Purpose Registers*) für allgemeine Aufgaben zur Verfügung. [1, S. 41] [2, S. 95]

Die *ALU* (Arithmetisch-Logische Einheit) der MSP430-CPU verarbeitet Operanden mit einer Breite von 16 Bit (bzw. bis zu 20 Bit bei der CPUX). Zu den unterstützten arithmetischen Operationen gehören Addition und Subtraktion (jeweils mit und ohne Übertrag) sowie die dezimale Addition mit Übertrag. Diese Operationen beeinflussen die Status-Flags (Overflow, Zero, Negative, Carry). Zudem beherrscht die ALU logische Operationen wie `AND` und `XOR`. Es ist zu beachten, dass die ALU – wie bei vielen Mikrocontrollern üblich – keine direkten Befehle für Multiplikation oder Division bereitstellt; diese müssen softwareseitig implementiert werden oder werden bei spezifischen Modellen durch einen Hardware-Multiplizierer unterstützt. [2, S. 95ff]

Die Controller der MSP430x2xx-Familie verfügen über einen 16-Bit-Adressraum, wodurch theoretisch bis zu `65.536` Speicherzellen (64 kByte) adressierbar sind. Der Speicher ist byteweise organisiert (8 Bit pro Zelle). Allerdings ist der physische Speicher nicht durchgehend belegt; es existieren Lücken im Adressbereich, die anwendungsspezifisch beachtet werden müssen. Je nach Derivat variiert die Speicherausstattung, um wirtschaftliche Lösungen für unterschiedliche Applikationsanforderungen zu ermöglichen. [1, S. 41]

Der Datenspeicher ist als *RAM* ausgeführt und erlaubt sowohl Lese- als auch Schreibzugriffe. Obwohl der Speicher byteweise organisiert ist, ermöglicht die 16-Bit-Architektur des MSP430 auch einen wortweisen Zugriff (16 Bit). In diesem Fall werden zwei adressseitig aufeinanderfolgende Speicherzellen quasi-simultan gelesen oder beschrieben. [1, S. 42]

3.1.2 | Eigenschaften verschiedener Derivate

Die MSP430-Familie beinhaltet verschiedene Derivate, welche sich beispielsweise bezüglich der integrierten Peripherie, der Taktrate und des Arbeitsspeichers unterscheiden. [3] Gründe für die hohe Vielzahl an Derivaten (laut Herstellerangaben zeitweise über 500 Varianten) [3] liegen in den spezifischen Anwendungsfällen. Zur Steigerung der Wirtschaftlichkeit soll vermieden werden, dass ungenutzte, kostenintensive Hardwarekomponenten im Endprodukt verbaut sind. [1, S. 41, S. 42ff] Die Integration dieser Peripheriemodule zielt primär darauf ab, die CPU von monotonen Routineaufgaben zu entlasten und gleichzeitig die Kommunikation mit der Systemumgebung zu gewährleisten. [1, S. 42ff]

Technisch betrachtet bilden alle Baugruppen, die an den Systembus angeschlossen sind (außer Speicher und CPU), das I/O-Subsystem. Hierzu zählen sowohl externe Ein- und Ausgabegeräte als auch interne Spezialregister zur Systemsteuerung. [2, S. 109ff] Die Einbindung dieser Komponenten erfolgt bei der MSP430-Familie über eine *Memory-Mapped-I/O-Architektur*. Da die Controller der Familie *MSP430x2xx* über einen gemeinsamen Adressraum von 64 kByte verfügen, sind sowohl Speicherzellen als auch Peripheriebausteine in diesem linearen Adressbereich angesiedelt. [1, S. 43] Die Peripherieschnittstellen fungieren dabei als Brücke zwischen dem externen Gerät und dem internen Bus. Jede Schnittstelle verfügt über Register für den Datenaustausch sowie für Status- und Steuerinformationen. Für die CPU stellt sich ein Peripheriegerät somit wie ein herkömmlicher Speicherzugriff dar. [2, S. 110]

Diese Register befinden sich spezifisch im Adressbereich zwischen `0010h` und `01FFh`, während die sogenannten *Special-Function-Register* (SFR) den Bereich ab `0000h` belegen. Aus dieser Architektur ergibt sich der Vorteil, dass für die Kommunikation mit der Außenwelt dieselben Befehle und Adressierungsarten verwendet werden können wie für interne Speicheroperationen. [1, S. 43]

Besonders kennzeichnend für ein Derivat ist die Art und Anzahl der verbauten Peripherie, die je nach Platz- und Kostenanforderungen variiert. So stehen zur Anbindung an die Außenwelt parallele sowie serielle Schnittstellen zur Verfügung. Letztere werden häufig durch *USARTs* (Universal Synchronous/Asynchronous Receiver/Transmitter) realisiert, die Protokolle wie *UART*, *SPI* oder *I²C* unterstützen. Ergänzend finden sich bei entsprechenden Derivaten USB-Schnittstellen. [1, S. 42ff][2, S. 109ff]

Eine zentrale Rolle spielen Timer-Baugruppen, die für das Messen von Zeitintervallen, die Generierung von Signalen (zum Beispiel Pulsweitenmodulation) oder das Auslösen periodischer Interrupts programmiert werden können. Als sicherheitsrelevantes Element ist zudem ein *Watchdog Timer* (WDT) implementiert. Dieser setzt das System zurück, sollte er nicht in definierten Abständen durch das Programm bedient werden, und verhindert so Systemstillstände. [2, S. 110][1, S. 42ff]

Bezüglich der Auswertung von Analogwerten stehen je nach Ausführung verschiedene *Analog-Digital-Wandler* (ADC) zur Verfügung, die wahlweise als 16-Bit *Sigma-Delta* oder *Slope-Wandler* ausgeführt sind. [3] Während ADCs kontinuierliche Umweltvariablen einlesen, ermöglichen *Digital-Analog-Wandler* (DAC) die Ausgabe analoger Signale. Einige Derivate verfügen zudem über integrierte Operationsverstärker zur Signalaufbereitung. [1, S. 43][2, S. 109ff] Zur

Unterstützung der Softwareentwicklung sind ferner Module integriert, die das Einspielen von Programmen und das Debugging (zum Beispiel via *Embedded Emulator*) ermöglichen. [2, S. 110]

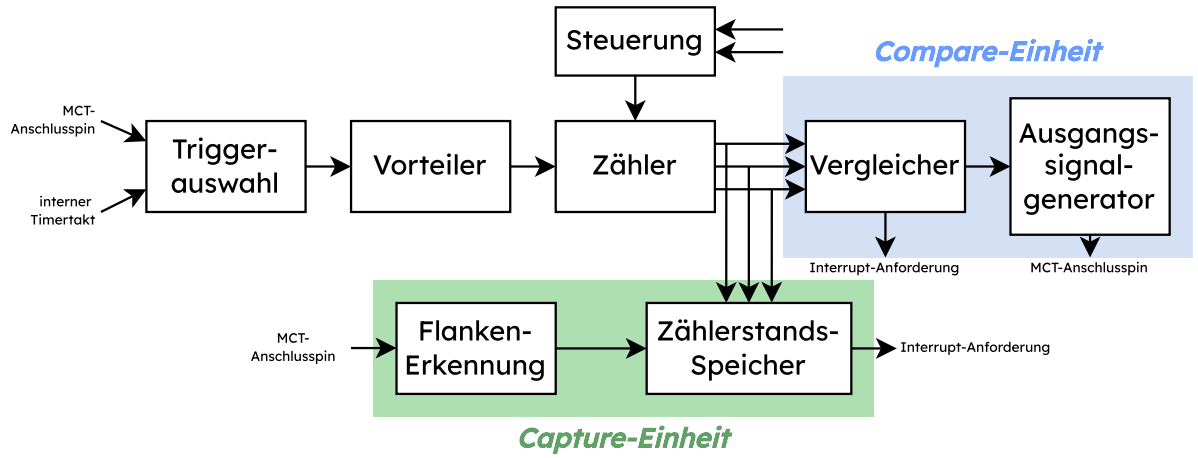


Abbildung 1: *Blockschema eines Timers mit Capture- und Compare-Einheit* [4, S. 600]

3.2 Capture-Compare-Einheit allgemein

In diesem Abschnitt wird die der späteren Simulation zugrunde liegende Funktionsweise der Capture-Compare-Einheit erläutert. Hierzu wird zunächst der grundlegende Aufbau und die Arbeitsweise der Timer betrachtet, auf denen sowohl der Capture- als auch der Compare-Modus basieren. Darauf aufbauend werden die Verarbeitung externer Signale sowie die Erzeugung von Ausgangssignalen beschrieben.

3.2.1 | Aufbau von Timern

Grundsätzlich besteht der Timer eines Mikrocontrollers aus einem Zähler mit einer Wortbreite n. Je nach Einstellung zählt dieser mit jedem Impuls abwärts oder aufwärts, wobei die Quelle dieser Impulse entweder durch eine interne Clock oder durch eine extern verbundene Taktquelle erzeugt wird (*Triggerauswahl*). Auch durchläuft das Taktsignal einen Vorteiler, durch dessen Einstellung die Zählfrequenz herabgesetzt werden kann. [4, S. 598f] [5, S. 169f] Wird bei jedem Maschinenzyklus der Speicherwert um 1 inkrementiert, so ergibt sich automatisch beim Wert 2^n ein periodischer Überlauf, der als *Timer-Flag* bezeichnet wird. Dem Progamm entsprechend kann dann ein Handling durchgeführt werden, das entweder in einer manuellen Abfrage oder einem automatischen Interrupt mündet. [6, S. 96] [5, S. 169] Das Setzen dieser Flag entspricht in der Digitaltechnik einem aktiven Carry- beziehungsweise Zero-Bit. Für die Steuerung des Timers

stehen dem Anwender verschiedene Register zur Verfügung, welche die Zählrichtung festlegen oder den Timer starten und stoppen. Ferner sind Register für die Interruptverwaltung sowie die Vorteiler- und Taktauswahl vorgesehen. [4, S. 599f]

Abbildung 1 zeigt den Aufbau des Timers. Die Capture- und Compare-Komponenten sind jeweils farblich hervorgehoben, sodass in den folgenden Abschnitten eine gesonderte Betrachtung dieser erfolgen kann.

3.2.2 | Handhaben externer Signale im Capture-Modus

Ziel des Capture-Modus ist ein möglichst genaues Messen von Zeitintervallen. Dazu wird zunächst ein externes Signal auf Flanken analysiert, das bei ausreichender Steilheit einen Zählerstandspeicher füllt. In diesen Speicher wird der aktuelle Wert des Zählers geladen, sodass der Wert zu einem späteren Zeitpunkt ausgelesen werden kann. Der Zähler läuft dabei unterbrechungsfrei weiter.

Durch die bekannte Taktfrequenz ergibt sich mit $T = \frac{1}{f}$ eine Taktzykluszeit, die dann mit dem gespeicherten Zählerstand multipliziert wird. Die Gesamtdauer $T_{gemessen} = t_{capture} - t_{start}$ ergibt sich somit aus $n_{capture} * \frac{1}{f} - n_{start} * \frac{1}{f}$, wobei n den Zählerstand repräsentiert. Je geringer das zeitliche Intervall zwischen eingetretenem Ereignis und einem Taktwechsel ist, desto genauer kann die Zeitmessung erfolgen. Entsprechend ergibt eine hohe Taktfrequenz eine höhere Genauigkeit. [5, S. 171] Der entsprechenden Verschnitt, der durch die Messungenauigkeit entsteht, ist in Abbildung 2 dargestellt.

3.2.3 | Erzeugen von Pulsmustern im Compare-Modus

Einen umgekehrten Betriebsmodus bietet der Compare-Modus, bei welchem ein Ausgangssignalgenerator ein Digitalsignal am Anschluss des Mikrocontrollers erzeugt. Die Compare-Einheit kann in einem Timer mehrere Male verbaut sein, jeweils mit eigenen Werten. Die Kernfunktion liegt zunächst im Abgleich des Zählerwertes mit einem Vergleichswert: Sind beide Werte identisch, kann ein Interrupt ausgelöst oder ein Ausgangssignal erzeugt werden. Letzterer Betriebsmodus (Output Compare) erfordert je nach Mikrocontroller unterschiedlich komplexe Schaltungen. [4, S. 598f]

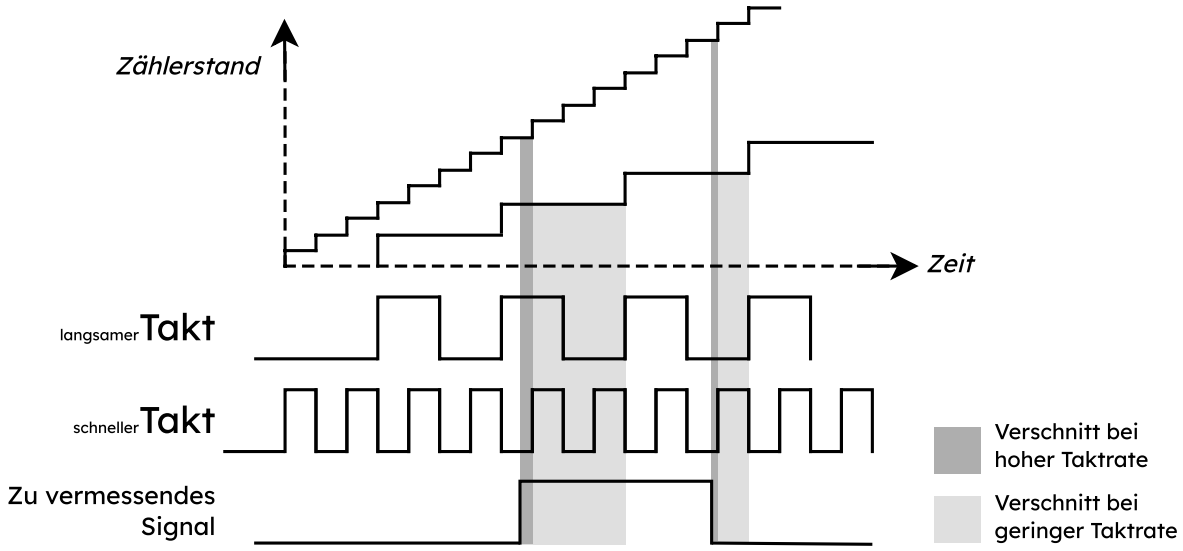


Abbildung 2: *Verschnitt im Capture-Modus: Je höher frequentiert sich der Zählerstand aktualisiert, desto direkter kann der Vergleich mit dem zu vermessenden Signal erfolgen* [5, S. 171]

Soll beispielsweise ein Rechtecksignal erzeugt werden, ist zunächst ein zu erreichender Zählwert zu definieren. Beim Timer des *STM32* von *STMicroelectronics* ist dieser im Register `TIMx_CCRy` abgelegt; [4, S. 611] Das zu erzeugende PWM-Signal soll nun bei jedem Erreichen dieses definierten Wertes umschalten. Für die Festlegung der Periodendauer des Rechtecksignals muss die gewählte Taktfrequenz mit dem eingestellten *Auto-Reload-Register* (ARR) abgeglichen werden. Bei einem 16 MHz-Timer entspricht jeder Takt einer Periodendauer von $1\ \mu s$. Mit einem eingestellten ARR von 999 wird der Endwert alle $1000\ \mu s = 1\ ms$ erreicht. Beim STM32 kann das Register CCR1 auf einen beliebigen Wert eingestellt werden, bei welchem der Mikrocontroller einen Toggle-Impuls ausführen soll. Durch die genannten Einstellungen entsteht der in Abbildung 3 gezeigte PWM-Verlauf. [4, S. 613]

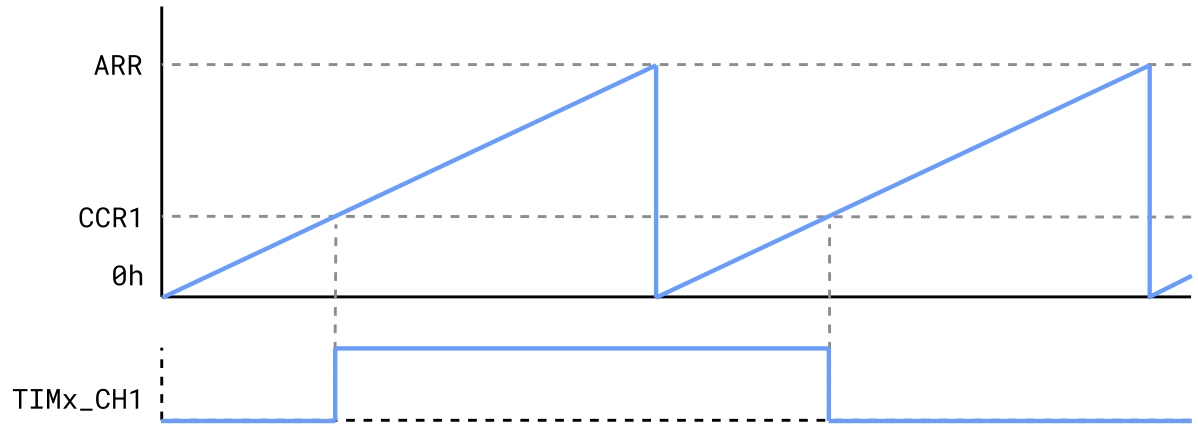


Abbildung 3: *Beispielhafte Einstellung des Compare-Registers beim STM32: Durch das Abbild von* `ARR` *und* `CCR1` *entsteht* `TIMx_CH1` [4, S. 613]

3.3 Timer_A im MSP430x5xx/x6xx

In diesem Kapitel wird explizit auf die Timer_A-Einheit und die Ouput-Modi der *MSP430x5xx*- und *MSP430x6xx*-Modelle eingegangen. Es wird sich eng an der Herstellerdokumentation orientiert, welche zuletzt die zu simulierenden Register enthält.

3.3.1 | Timer_A

Der für diese Arbeit zu skizzierende Timer_A ist ein 16-bit-Timer, welcher bis zu sieben Capture/Compare-Register bereitstellt. Gezählt wird entsprechend der am Eingang auftretenden Taktflanken. Der Entwickler kann zwischen vier verschiedenen Taktquellen wählen, die wahlweise über einen Vorteiler verlangsamt werden können. Abbildung 4 zeigt das Blockschaltbild des Timer_A und eine vereinfachte Darstellung einer generischen Compare-Einheit n. [1, S. 126]

Darüber hinaus bietet der Timer vier verschiedene Modi, mit denen sich das Zählverhalten anpassen lässt. Die vier Betriebsmodi lauten *Continuous Mode*, *Up Mode*, *Up/down Mode* und *Stop Mode*. [7, S. 463] Die Modi sind wie folgt definiert:

- Im *Continuous Mode* wird vom aktuellen Zählerstand aufwärts gezählt bis zum Maximalwert `0xFFFF`. Eine darauffolgend eintreffende Flanke führt zum Rücksetzen des Zählers auf den Wert `0x0000`. [7, S. 465]

- Der *Up Mode* unterscheidet sich vom Continuous Mode insofern, dass nicht bis zum Wert `0xFFFF` gezählt wird, sondern der Timer-Reset bereits bei Erreichen des vom Entwickler festgelegten `CCR0` -Werts erfolgt. [7, S. 464]
- In der Betriebsart des *Up/Down Mode* wird ebenfalls bis zum definierten `CCR0` -Register gezählt. Jedoch springt der Timer nicht mit einem geraden Abschluss zum Wert `0x0000` zurück, sondern zählt bei Erreichen von `CCR0` mit umgekehrter Zählrichtung abwärts. Insbesondere in diesem Modus bietet sich das Erstellen von symmetrischen PWM-Signalen an. [7, S. 466ff]

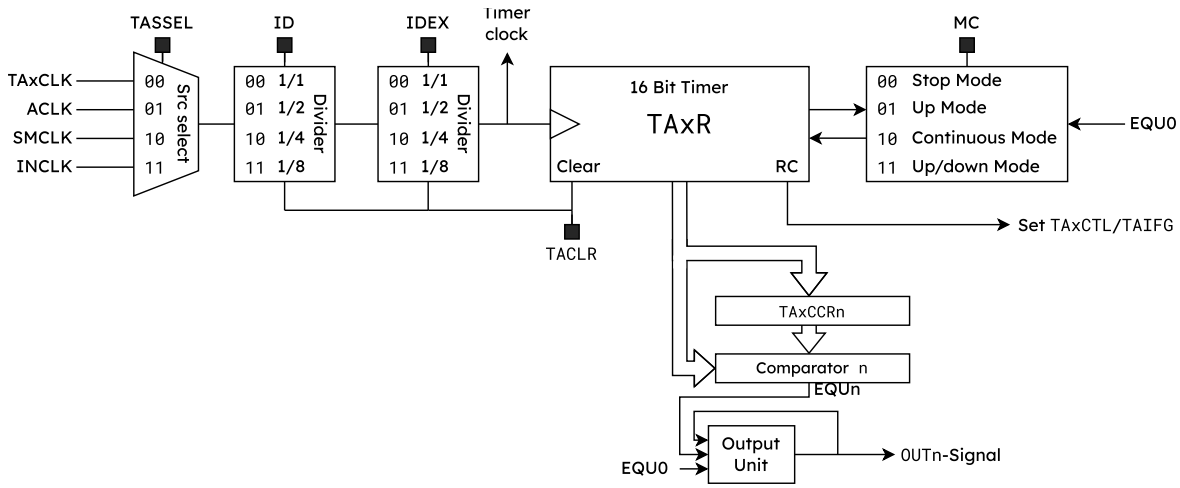


Abbildung 4: *Blockschaltbild des Timer_A der MSP430x5xx- und MSP430x6xx-Modelle mit Compare-Einheit* [7, S. 462]

3.3.2 | Output-Modi für die Compare-Einheit

Im Compare-Modus wird mit den vom Timer ermittelten Flags ein pulsweitenmoduliertes Signal erzeugt. Dazu stellt Texas Instruments acht verschiedene Output Modi zur Verfügung, die jeweils eine gesetzte Flag unterschiedlich verarbeiten und das Ausgangssignal manipulieren. Dazu ist das Einstellen von weiteren `TAxCCRn` -Werten möglich, welche individuelle Interrupts erzeugen, sobald Zählerwert und `TAxCCRn` -Wert äquivalent sind. Diese internen Signale werden in der Dokumentation des MSP430 als `EQUn` bezeichnet, wobei `n` für die entsprechende Capture-Einheit steht. [7, S. 469] Weil die *MSP430x5xx*- und *MSP430x6xx*-Familie bis zu sieben Compare-Units bereitstellen, ergibt sich die Möglichkeit des Setzens von maximal sieben Interrupts. [7, S. 475] [1, S. 132] Diese Modi sind in `Tabelle 1` aufgelistet.

In den Herstellerunterlagen wird das Schaltverhalten der Output-Modi zudem detailliert in grafischer Form dargestellt (vgl. Abbildung 5). Dabei wird unterschieden, ob sich der Timer bereits im eingeschwungenen Zustand befindet (gezeigt als ⎍) oder ob eine alternative Startbedingung vorliegt (⋮). Letzteres zeigt den Signalpfad, falls der Ausgangspegel zu Beginn des Zyklus nicht dem Endwert des vorangegangenen Zyklus entspricht.

Tabelle 1: *Output-Modi der MSP430x5xx- und MSP430x6xx-Modelle (Tabelle)* [7, S. 469]

OUTMODx	Mode	Description
000	Output	The output signal OUTn is defined by the OUT bit. The OUTn signal updates immediately when OUT is updated.
001	Set	The output is set when the timer *counts* to the TAxCCRn value. It remains set until a reset of the timer, or until another output mode is selected and affects the output.
010	Toggle/Reset	The output is toggled when the timer *counts* to the TAxCCRn value. It is reset when the timer *counts* to the TAxCCR0 value.
011	Set/Reset	The output is set when the timer *counts* to the TAxCCRn value. It is reset when the timer counts to the TAxCCR0 value.
100	Toggle	The output is toggled when the timer *counts* to the TAxCCRn value. The output period is double the timer period.
101	Reset	The output is reset when the timer *counts* to the TAxCCRn value. It remains reset until another output mode is selected and affects the output.
110	Toggle/Set	The output is toggled when the timer *counts* to the TAxCCRn value. It is set when the timer *counts* to the TAxCCR0 value.
111	Reset/Set	The output is reset when the timer *counts* to the TAxCCRn value. It is set when the timer *counts* to the TAxCCR0 value.

Bei der Analyse der Compare-Modi des Timer_A fällt im Up/Down-Modus eine Diskrepanz in der Dokumentation des Herstellers auf. Während die textuelle Definition (vgl. Tabelle 1) beispielsweise des Output Mode 2 (Toggle/Reset) einen Reset des Ausgangssignal beim Erreichen von `CCR0` vorsieht (»*It is reset when the timer counts to the TAxCCR0 value*«), [7, S. 469] zeigt das zugehörige Timing-Diagramm für den Up/Down-Mode in den Grenzen von `CCRn` bis `CCRn` einen durchgehenden `HIGH`-Pegel, sofern der Timer mit dem logischen Wert `FALSE` beginnt (vergleiche Abbildung 5). [7, S. 472] Auch der Zusatz, dass beispielsweise im Modus 7 (Reset/Set) der Reset nur bei fallender Timer-Flanke auslöst (so suggeriert es die Grafik), bleibt in der Beschreibung unerwähnt. [7, S. 469, 472] Diese Ambivalenz der Dokumentation macht eine empirische Validierung für die Erstellung eines zyklusgenauen Simulators notwendig. Abbildung 6 zeigt eine direkte Gegenüberstellung der textuellen Beschreibung (links) des Schaltverhaltens mit den von TI in der Dokumentation angegebenen Grafiken (rechts). Die Unterschiede sind jeweils rot markiert. Zusätzlich suggeriert die

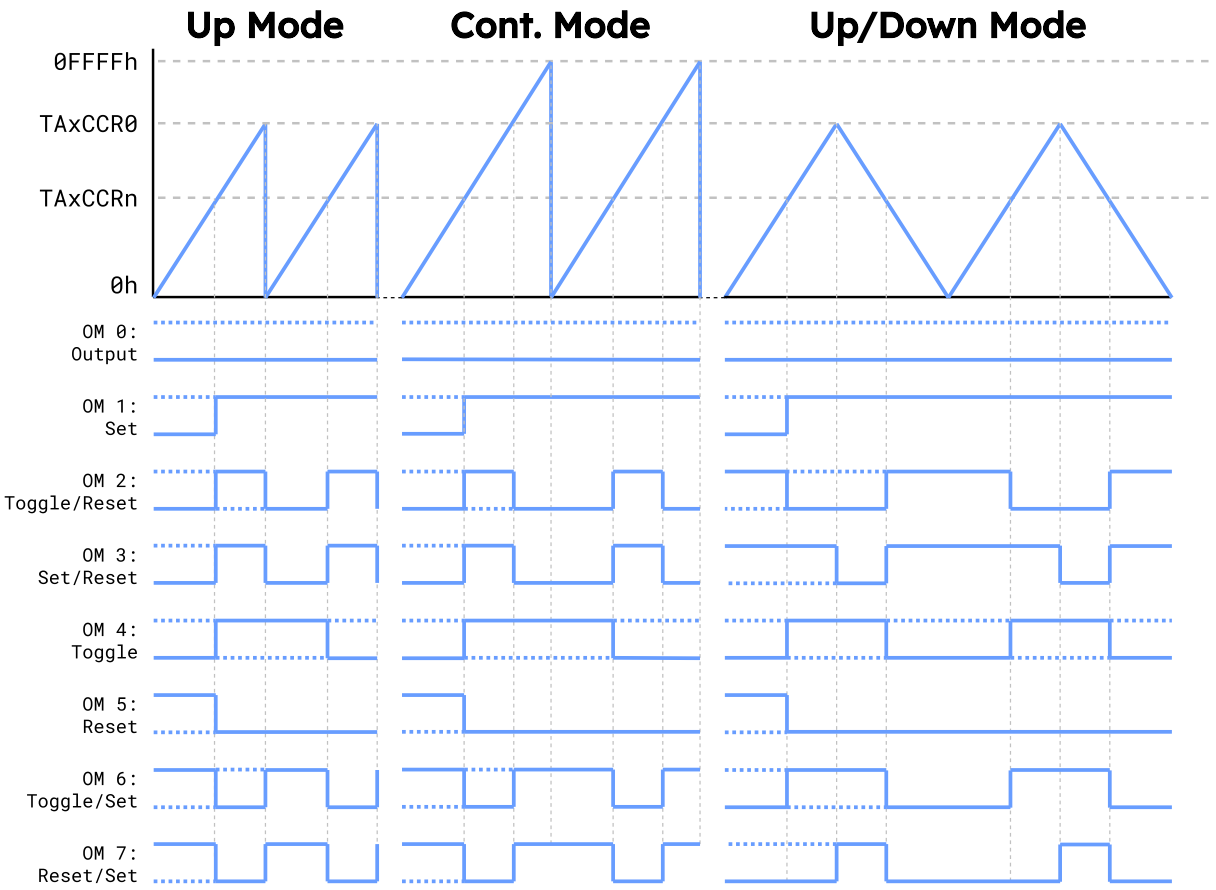


Abbildung 5: *Output-Modi der MSP430x5xx- und MSP430x6xx-Modelle (Grafik); kompakte Darstellung von* [7, S. 470-472]

grafische Erklärung, ein Set oder Reset wäre abhängig von der Zählrichtung des Timers, da andernfalls ein Verhalten wie im Output Modus 2, 3, 6 und 7 nicht erklärbar wären. Daher sind die Stellimpulse in Abbildung 6 teilweise mit der zugehörigen Bedingung an die Zählrichtung des Timers durch einen Pfeil gekennzeichnet ($\uparrow_n$). Das tiefgestellte n zeigt zusätzlich, dass mit dem Stellimpuls `CCRn` gemeint ist. Zuletzt deutet ein ⚠-Symbol darauf hin, dass die grafische Beschreibung dem eigentlichen Modus-Namen direkt widerspricht. Das betrifft Modus 2 und 7, bei denen ein Set-Impuls bei Erreichen von `CCRn` zum Reset führt und umgekehrt.

Weiterhin bildet `CCR0` eine Ausnahme: Läuft der Timer, fällt die `EQUn`-Flag mit der `EQU0`-Flag aufeinander. Wird ein Modus gewählt, welcher unterschiedliche Aktionen erfordert, sobald `CCR0` und `CCRn` erreicht werden, ergibt sich ein Widerspruch. So sieht *Modus 3 (Set/Reset)* ein *Setzen* vor, sobald `CCRn` erreicht wird. Jedoch soll das Signal auch rückgesetzt werden, wenn der Timer-Wert `CCR0` entspricht – in der Theorie werden zum gleichen Zeitpunkt

zwei Umschaltvorgänge aktiviert. Dies bildet eine Besonderheit im Vergleich zu den Registern `TAxCCR1` - `TAxCCR6`; für die spätere Simulation ist es ungewiss, welchem der Befehle im Halbleiter Vorrang eingeräumt wird. Entsprechend würde das Ausgangssignal dauerhaft im logischen `LOW` verweilen, sollte zum Beispiel der Reset überwiegen. Andernfalls könnten am Ausgangssignal Nadelimpulse auftreten, wenn das Rücksetzen im Halbleiter geringfügig länger dauerte als das Setzen.

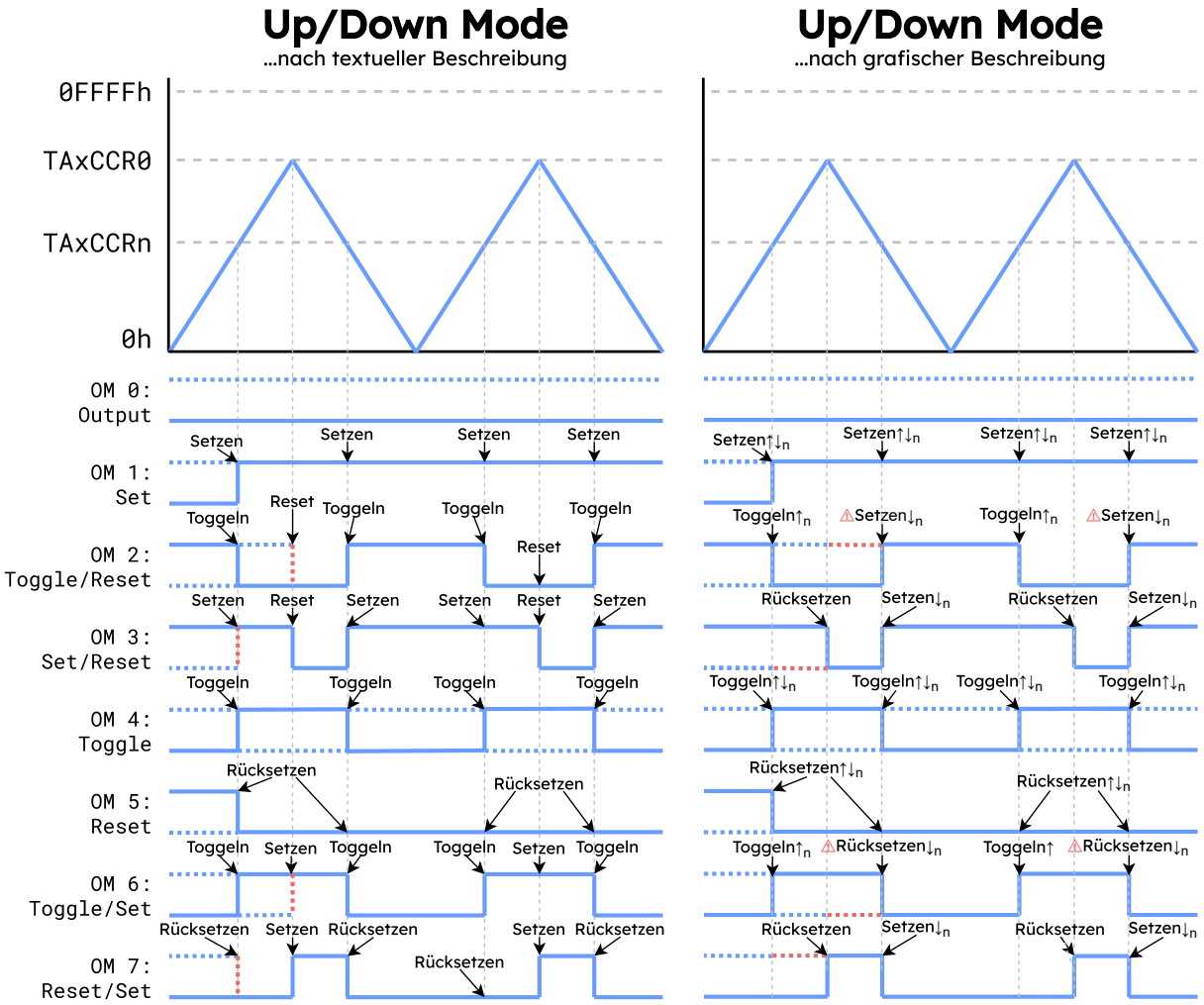


Abbildung 6: *Diskrepanzen des dokumentierten Up/Down-Mode der MSP430x5xx- und MSP430x6xx-Modelle; kompakte Darstellung von [7, S. 470-472] (rechts) und textueller Erklärung von [7, S. 469] (links). Die Unterschiede sind jeweils rot markiert. Scheint ein Set oder Reset abhängig von der Zählrichtung (Flanke) zu sein, ist dies mit einem Pfeil-Symbol (↑n) gekennzeichnet. Das ⚠-Symbol deutet darauf hin, dass die grafische Beschreibung dem eigentlichen Modus-Namen direkt widerspricht.*

3.3.3 | Kontrollregister und Makros

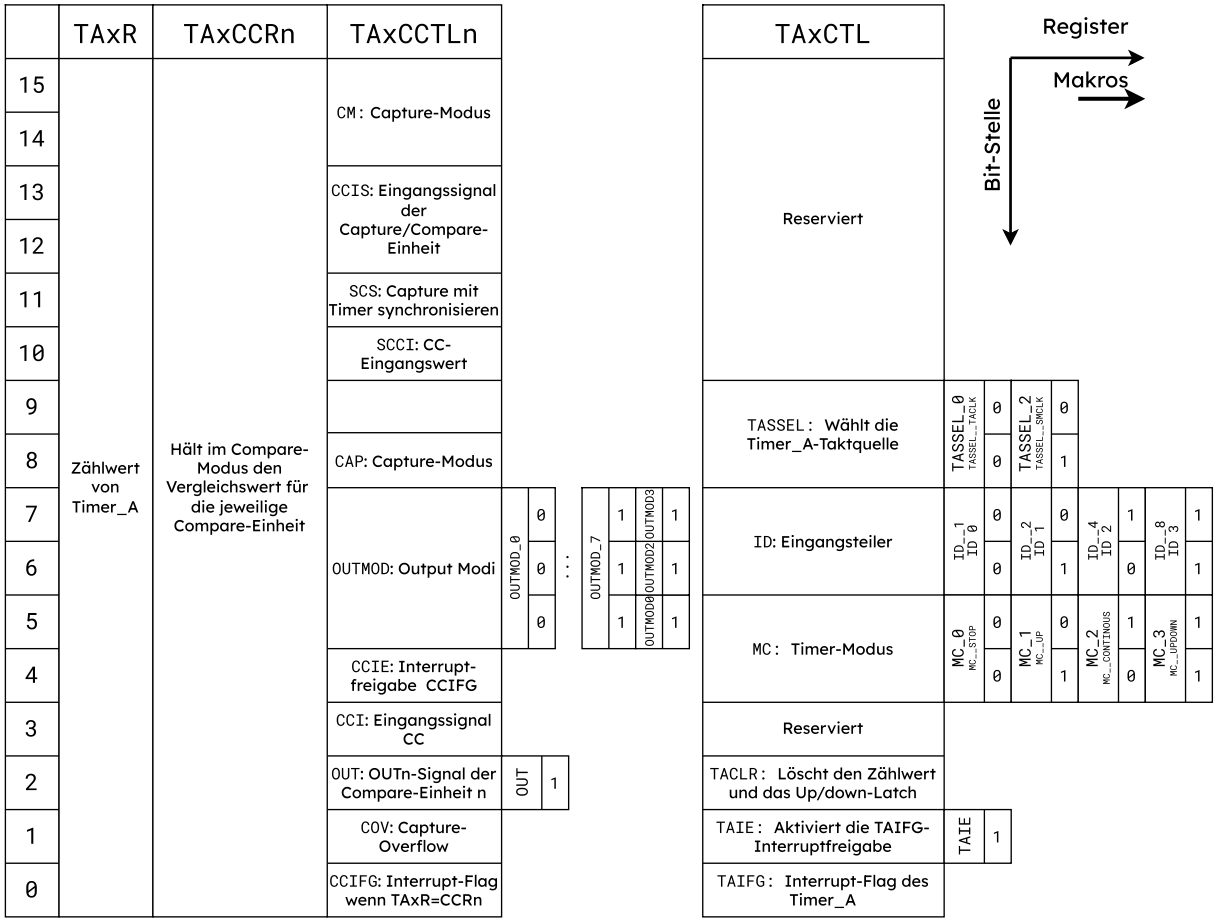
Zur Steuerung der Timers sowie der Capture-/Compare-Einheit werden diverse Register verwendet. Nicht alle davon sind für diese Arbeit von Bedeutung, weshalb diese nicht weiter erläutert werden. Die im Folgenden maßgeblichen Register sind in Tabelle 2 aufgezählt, inklusive einer Auswahl der von Texas Instruments festgelegten Makros, wie sie etwa im *gpp-Compiler* oder offiziellen *TI Code Composer Studio* enthalten sind. [8]

Zunächst existiert das Zählregister `TAxR`, das den aktuellen Wert des Timers enthält. Es ist – ebenso wie die anderen Register – 16 Bit breit. Die Register `TAxCCRn` beinhalten ebenfalls Zählwerte, jedoch jene, mit denen der Timer-Wert verglichen werden soll. [7, S. 470-472]

Die tatsächlichen Kontrollregister lauten `TAxCCTLn` und `TAxCTL`. Das übergreifende `TAxCTL`-Register steuert das Verhalten des Timers, weshalb dort Einstellungen wie die Wahl der Taktquelle, des Eingangsteilers oder des Timer-Modus getroffen werden können. Ferner existiert die Interruptfreigabe `TAIE` mit zugehöriger `TAIFG`-Flag. Diese `TAIFG`-Flag wird im Takt gesetzt, bei welchem der Timer vom Wert `CCR0` zu `0` zurückkehrt. Durch Setzen des Bit 2 (`TACLR`) kann zudem der Zählwert manuell zurücksetzt werden; weil im Up/down-Modus ein internes Latch die aufsteigende oder absteigende Zählrichtung koordiniert, wird dieses Latch ebenfalls durch `TACLR` zurückgesetzt, sodass initial aufwärts gezählt wird, wenn der Timer erneut startet.

Die mehrfach vorhandenen `TAxCCTLn`-Register beziehen sich jeweils auf eine Capture/Compare-Einheit und beinhalten Kontroll-Bits für die Capture-Einheit, die Interruptfreigabe und die Output-Modi. Für die Compare-Unit sind dabei `OUTMOD` (Toggle, Set/Reset, usw...) und `OUT` (Ausgangs-PWM-Signal, global als `OUTn`) relevant.

Tabelle 2: *Übersicht der Register zur Steuerung der Capture-Compare-Unit des MSP430x5xx/x6xx. Alle mit einem Makro genannten Bits sind für die spätere Compare-Simulation relevant.* [7, S. 476-480]



3.4 Einführung in die Webprogrammierung

Zum Abschluss der theoretischen Grundlagen wird in diesem Kapitel die Methodik zur Erstellung von Webseiten erläutert, welche im darauffolgenden Hauptkapitel die Basis für die Umsetzung des Demonstrators bildet.

3.4.1 | HTML und CSS

HTML und CSS sind die Grundbausteine, welche die Inhalte einer Webseite festlegen und formatieren. HTML legt zunächst die einzelnen Komponenten der Seite fest und definiert die allgemeine Struktur. Dazu werden Tags verwendet, welche den gewünschten Inhalt umfassen. So erstellt beispielsweise

`<h1>Überschrift</h1>` eine mit der Standardformatierung des Browsers versehene Überschrift (Header). [9, S. 8] Zusätzlich können Tags spezielle Attribute enthalten, welche das Element mit Steueranweisungen versehen. Ein typisches Beispiel stellt dabei der *Link-Tag (Anchor-Tag)* dar, der über das Attribut auf die Zielressource verweist: `<a href="https://google.com/">Google öffnen</a>`. [9, S. 20] Auch die später verwendeten Eingabelemente verwenden diese Attribute, um den Typ eines Eingabefelds oder den gewährten Wertebereich zu definieren. [9, S. 30] `<input id="CCR0" type="slider" max="10" min="1" value ="5" />"` erzeugt im Browser einen Schieberegler, welcher die Werte 1 bis 10 zulässt und standardmäßig bei Seitenaufruf den Wert 5 vorgibt. Das `id`-Attribut kann schließlich verwendet werden, um per JavaScript dieses Feld gezielt auszulesen.

Die genannten HTML-Tags können nun als *Selektoren* angesprochen werden, um mit CSS deren Formatierung zu ändern. Der einfachste CSS-Selektor ist der Tag selbst, also `h1 { color: red;}` um die Überschrift `h1` mit rotem Text darzustellen. Da es auf einer Webseite aber meist mehrfach verwendete Tags gibt, welche jedoch unterschiedlich zu formatieren sind, muss der Selektor entsprechend präzisiert werden. Zwei typische Selektoren sind der Klassen-Selektor (`class`) und Individualformate (`id`). Dass ein Element einer Klasse zugeordnet werden soll, wird über die Schlüsselworte `class="..."` festgelegt; bei einer ID analog `id="..."`. Der CSS-Selektor lautet dann nicht mehr `h1 {...}`, sondern für eine Klasse `.titel { color: red;}` bzw. `#titel { color: red;}` für ein Individualformat. [9, S. 48ff]

3.4.2 | JavaScript-Canvas

Ergänzend zu HTML und CSS sorgt JavaScript für die entsprechende Dynamik der Webseite. Das meiste von dem, was nicht statisch angezeigt wird (abgesehen von CSS-Animationen), wird durch JS erzeugt. Insbesondere Interaktionen mit dem Nutzer oder die Auswertung von Eingaben obliegen JS. JavaScript entstand in den 1990er Jahren aus dem von der Firma Netscape entwickelten LiveScript; aufgrund der zu dieser Zeit hohen Beliebtheit von Java wurde LiveScript an die Syntax von Java angelehnt, wenngleich sie keinesfalls die gleiche Programmiersprache sind. Später wurde schließlich aus LiveScript das heute bekannte JavaScript. [10, S. xxi]

Neben den Grundprinzipien von Berechnungen, Variablen, Funktionen oder Objekten bietet JS den für diese Arbeit besonders relevanten *Canvas-Bereich*, welcher für die (animierte) Bilderzeugung genutzt wird. [10, S. 6] Weil ein JS-Canvas

pixelorientiert ist, fußt sowohl die Definition der Canvas-Größe als auch die absolute Positionierung von Elementen innerhalb des Canvas auf der Angabe eines Pixel-Koordinatensystems. Der Ursprung liegt oben links. Für das konkrete Zeichnen des Canvas muss zunächst die entsprechende Fläche (Höhe und Breite) definiert werden. Dies geschieht durch den entsprechenden HTML-Tag `<canvas id="canvas" width="1080" height="720"></canvas>`. In JS selbst muss dann der Canvas mit `var canvas = document.getElementById("canvas");` in eine Variablen-Referenz (hier `canvas`) geladen werden und anschließend der Rendering-Context (hier 2-Dimensional) durch `var ctx = canvas.getContext('2d');` festgelegt werden. [10, S. 111ff] Anschließend kann durch die Befehle

```
ctx.moveTo(0, 0);
ctx.lineTo(200, 100);
ctx.stroke();
```

eine Linie von der Koordinate (0, 0) nach (200, 100) gezeichnet werden.

4 Umsetzung

In diesem Kapitel wird die praktische Umsetzung der zuvor dargestellten theoretischen Konzepte beschrieben. Ziel ist es, die Funktionsweise des Timer_A des MSP430 sowie dessen Interrupt- und Ausgangsverhalten in einer webbasierten Anwendung abzubilden und nachvollziehbar darzustellen. Dabei werden sowohl konzeptionelle als auch technische Aspekte der Realisierung betrachtet.

Zu Beginn werden grundlegende Vorüberlegungen zur Umsetzung erläutert, die den strukturellen Rahmen der Anwendung sowie den allgemeinen Ablauf definieren. Diese bilden die Grundlage für die nachfolgenden Abschnitte, in denen die einzelnen funktionalen Bestandteile detailliert beschrieben werden.

Hinweis: Die im Folgenden gezeigten Abbildungen von Timern sind Reproduktionen, die in externer Software erstellt wurden, um eine hohe Darstellungsqualität ohne Bildrasterung zu gewährleisten.

4.1 Vorbetrachtungen zur Umsetzung

Zunächst dient dieses Kapitel einer kurzen Vorbetrachtung, die das Layout und den übergreifenden Ablauf beschreibt. Da die Simulation in mehreren Schritten die Ausgangssignale ermittelt, wird ein für den gesamten Praxisteil der Arbeit gültiger Ablaufplan erstellt, welcher der Orientierung dient.

4.1.1 | Layout und Einbettung

Der Foliensatz des Mikrocontroller-Labors ist mit der JavaScript-Bibliothek *Reveal.js* umgesetzt. Die einzelnen Folien befinden sich im Rahmen eines `<section>...</section>`-Elements, das für die aktuell zu zeigende Folie den Bildschirminhalt füllt. Dabei ist jedoch auf die Abmessungen der zu präsentierenden Inhalte zu achten, da es andernfalls zu unerwünschten Anzeigefehlern kommt. Abbildung 7 zeigt eine Bildschirmaufnahme

einer Folie, bei welcher absichtlich ein zu hoher Inhalt hinzugefügt wurde. Auf Endgeräten mit Breitbildformat kommt es zu einem Textüberfluss über die Bildschirmunterkante hinaus. Umgekehrt wahrt *Reveal.js* stets stets die maximale Seitenbreite und skaliert die Folie so, dass sie von der Breite die gesamte Fensterbreite einnimmt. Entsprechend soll der zu entwickelnde Demonstrator ebenfalls das Breitbildformat nutzen, um Darstellungsfehler zu verhindern.

Abbildung 7: *Bildschirmaufnahme einer Folie des Mikrocontrollerlabors mit einem zu hohem Inhalt: Reveal.js skaliert einen vertikalen Überlauf nicht nach, sondern lässt es zum Textüberfluss an der unteren Bildschirmkante kommen. Ein Scrollen ist nicht möglich.*

Ziel ist, dass der Demonstrator etwa 60% der Bildschirmbreite einnimmt, während die Bedienoberfläche die restlichen 40% abdeckt. Die abschließende Einbettung kann dann im gewünschten Format in Fort eines `<iframe>...</iframe>`-Elements erfolgen.

4.1.2 | Übergreifender Ablauf

Die Verarbeitung von der Eingabe der Werte bis zur Darstellung der `OUT`-Signale folgt einem sequenziellen Ablauf mit aufeinander aufbauenden Zwischenschritten, wie in Abbildung 8 gezeigt.

Es gibt dabei zwei Variablen, welche für die Weitergabe der Werte von besonderer Wichtigkeit sind. `graphIntersections[]` speichert zunächst alle Schnittpunkte, bei welchem ein CCRn-Wert dem Timer-Wert entspricht. Zusätzlich ist hier die `TAIFG`-Flag gespeichert, da dies technisch einem Schnittpunkt mit der X-Achse entspricht. Das Array `outputLevels[]` beinhaltet anschließend den letztlichen Ausgangspegel.

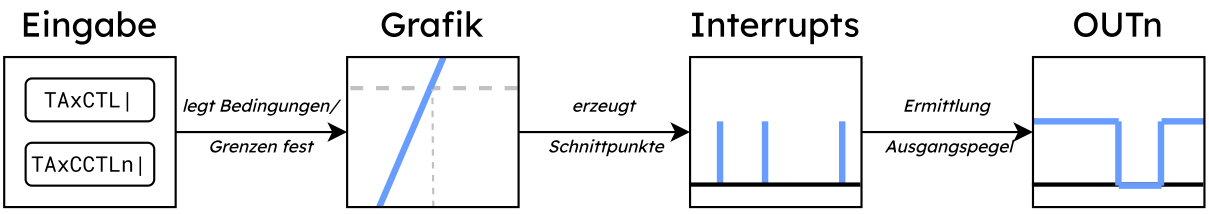


Abbildung 8: *Ablaufbasierte Darstellung des Demonstrators. Hinweis: Die Grafik ist klickbar und führt zum entsprechenden Kapitel.*

4.2 Verarbeitung der Nutzerdaten

Dem Nutzer wird erlaubt, alle Eingaben im binären, hexadezimalen oder dezimalen Format vorzunehmen. Außerdem sind die von TI definierten Makros gemäß `msp430f5529.h` in Teilen verwendbar. Zusätzlich konvertiert das Programm die Eingaben stets in die übrigen Zahlenformate.
Die Eingabefelder verfügen zunächst über eine einfache Texteingabe, sodass die gewünschten Werte entgegengenommen werden. Für eine komfortablere Einstellung der `CCRn` -Grenzen besteht darüber hinaus die Möglichkeit, über einen Schieberegler Werte vorzugeben. In beiden Fällen wird bei einer geänderten Eingabe stets eine `updateValueDisplay()`-Funktion aufgerufen, welche einige Prüfungen durchführt (gegebenenfalls auch Bits isoliert) und die für die Anzeige wichtigen Werte in versteckte HTML-Felder zurückschreibt.

Die zuvor genannte Funktion `updateValueDisplay(type, source, standanalone)` erhält zunächst den Registernamen (*type*), das aufrufende Element (*source*) und einen Boolschen Wert, ob die Eingabe in ein anderes Eingabefeld zurückgeschrieben werden soll (*standalone*). Alle Parameter werden vom entsprechenden `onchange=""` mitgeliefert. `type` und `source` sind für die Zuordnung der Eingaben bedeutsam. `standalone` verhindert ein »Glitching«-Verhalten, bei welchem der zuvor eingegebene Wert unmittelbar über die Funktion wieder zurückgeschrieben würde. Wurde der Wert aber über den Schieberegler angepasst, soll nur das Textfeld mit Inhalt gefüllt werden und andersherum.

Für `TAxCCTLn` und `TAxCTL` werden zusätzlich Makros akzeptiert, die ebenfalls verarbeitet werden müssen. Grundsätzlich sind alle unterstützten Werte in Konstanten gespeichert, die über die im Code verwendete Funktion `eval()` direkt vom Nutzer genutzt werden können. Zur Vereinfachung der Bedienung erfolgt jedoch eine Vorauswahl der Werte, welche auf ungeeignete und von der Simulation nicht unterstützte Eingaben hinweisen soll. Daher ist (neben den Konstanten) ein Array definiert, das jeweils

»verbotene« Kontroll-Bits beinhaltet. Dieses Array unterscheidet weiterhin zwischen verwendeten Registern, sodass die Hinweise nur bei den tatsächlich sinnvollen Registern erscheinen. Einen Ausschnitt dieses Arrays bietet `Codeblock 1`.

```
const incompatibleBitValueMap = {
  "TASSEL_1":       [0x0100, ["TAxCTL"]],
  "TASSEL_3":       [0x0300, ["TAxCTL"]],
  "CM_0":           [0x0000, ["TAxCCTL0","TAxCCTL1","TAxCCTL2"]],
  "CM_1":           [0x4000, ["TAxCCTL0","TAxCCTL1","TAxCCTL2"]],
  "CM_2":           [0x8000, ["TAxCCTL0","TAxCCTL1","TAxCCTL2"]]
};
```

Codeblock 1: *Array mit explizit als inkompatibel markierten Registern (Auszug)*

Zur Fehlerbehandlung werden die Eingaben auf mehreren Ebenen überprüft, wobei die zuerst durchgeführte Prüfung unmittelbar einen dem jeweiligen Register zugeordneten Fehlerspeicher beschreibt. Zur Verdeutlichung dienen die folgenden Beispiele:

- Die Eingabe `TASSEL_2 | MC_1` kann korrekt gelesen werden. Die Operation wird mit `eval()` ausgeführt.
- Eine Eingabe `TASSEL_2 | CM_2` im `TAxCTL`-Register enthält das dem Register unbekannte `CM_2`-Makro, weshalb die Meldung »*Die Anweisung(en) 'CM_2' verstehe ich nicht!*« erscheint.
- Eine Eingabe `OUTMOD_2 | CM_2` im `TAxCCTL0`-Register wiederum enthält das *grundsätzlich* nicht unterstützte `CM_2`-Makro, weshalb die Meldung »*Wird in dieser Simulation nicht unterstützt!*« angezeigt wird.
- Die fehlerhafte Eingabe `OUTMOD_3 (` kann aufgrund des Klammerfehlers nicht ausgeführt werden, weshalb dies im JavaScript-Fehler »*expected expression, got end of script*« mündet.

Wichtig ist, dass die Menge der nicht unterstützten Makros und die Menge der unbekannten Makros keine Schnittmenge bilden, und das unabhängig vom Register. Dies bedeutet, dass ein Register ein Makro als unbekannt einstufen kann, während die Verarbeitung (und damit die spezifische Fehlerausgabe) einer nicht unterstützten Eingabe allein durch das Register erfolgt, das dieses Makro in seinem Umfang kennt.

Sofern alle Eingaben korrekt verarbeitet und die Makros verwendet wurden, wird der Wert mit `.padStart()` in alle Zahlenformate umgewandelt und ausgegeben. Die einzelnen, für das Register relevanten Bitstellen, werden isoliert und in versteckte HTML-Input-Elemente geschrieben. Zuletzt erfolgt eine weitere Prüfung anhand des `incompatibleBitValueMap`-Arrays, da auch über numerische Eingaben nicht erlaubte Bits

angesprochen werden können. Dies erfolgt jedoch weniger präzise, weil es für den `TAxCTL`-Wert `0x0100` eine Doppelbelegung gibt (`TASSEL_1` und `TASSEL__ACLK`). Diese Aliase werden nicht herausgefiltert. Jedoch dient auch hier das innere Register-Array der Spezifizierung der Register. Deshalb führt die Eingabe von `0x0001` im `TAxCTL`-Register zur Ausgabe, dass `TAIFG` nicht untersützt ist, während die Fehlerausgabe von `TAxCTL` die unerlaubte `CCIFG`-Verwendung bemängelt. Beispiele für die interpretierten Eingabewerte bietet Abbildung 9.

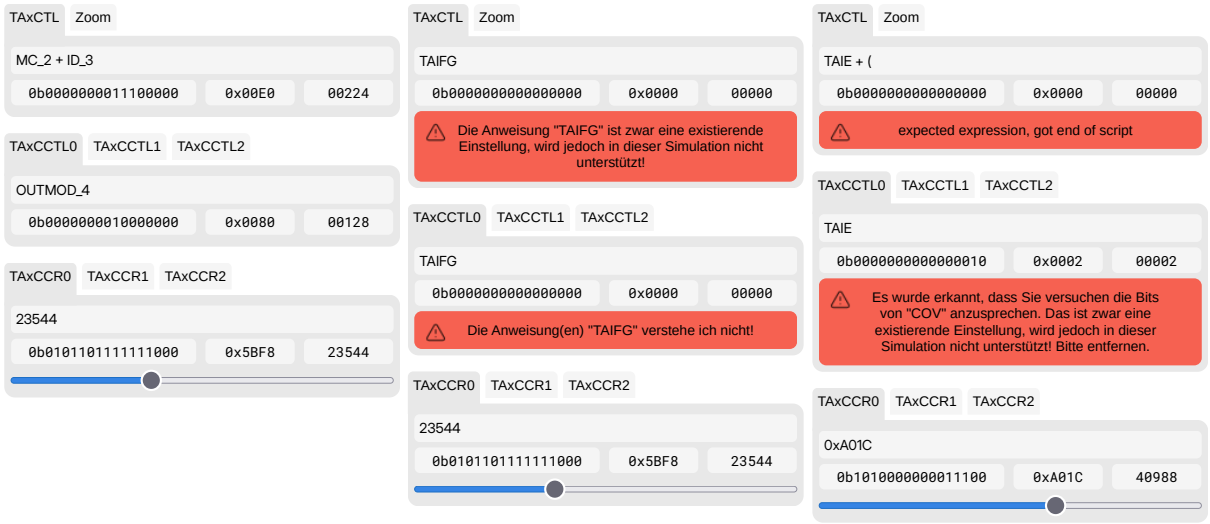


Abbildung 9: *Eingabemasken mit verschiedenen Eingabewerten: Makros werden nur im zugehören Eingabefeld als inkompatibel bemängelt (Mitte). Trifft ein Makro auf eine nicht unterstützte Bitstelle zu, wird dies ebenfalls erkannt (rechts).*

4.3 Darstellung des Timers und Interrupts

Dieses Kapitel befasst sich mit der grafischen Darstellung des Timers sowie der Ermittlung der daraus resultierenden Interrupt-Ereignissen. Der zeitliche Verlauf des Timerzählers wird modelliert und in eine visuelle Form überführt. Dabei werden die relevanten Schnittpunkte zwischen Zählerstand und Compare-Werten berechnet und für die weitere Verarbeitung aufbereitet.

4.3.1 | Abbildung des Timers

Der Timer wird in einem Zeit-Wert-Koordinatensystem angezeigt, wie auch in der Dokumentation von TI. Dazu befindet sich auf der linken Seite der X-Achse eine

Beschriftung der Werte von `CCRn` sowie eine Angabe der maximalen und minimalen Werte (`0x0000` und `0xFFFF`). Die Labels folgen jeweils der eingegebenen Werte. Auf gleicher Höhe sind dann die Werte durch durchgehende Strichlinien innerhalb des Koordinatensystems dargestellt. Die Box des Koordinatensystems ist mit 10px zur Bildschirmoberkante sowie mit 110px zur linken Kante gezeichnet. Die Abmessungen betragen 200x800px (Höhe x Breite).

Grundlage für das Zeichnen des Timers ist eine Schleife, welche fortlaufend ausgeführt wird, bis eine Breite von 1600 Pixeln erreicht ist (also der Graph die gesamte Breite füllt und darüber hinaus). Die Schleife ist nicht dazu gedacht, einen tatsächlichen Zählwert zu inkrementieren/dekrementieren, sondern zeichnet kontinuierlich Rampen-Pfade, welche insgesamt die Breite von 1600 Pixel annehmen sollen. Weil je nach eingestellter Zoomstufe sehr viele Impulse dargestellt werden, verhindert eine Laufvariable eine übermäßig häufige Ausführung der Schleife, um einen Absturz des Browsers zu verhindern. Es wird zudem festlegt, bis zu welchem Wert der Timer zählen soll. Im Continous Mode ist dies `0xFFFF` , in den übrigen Fällen `CCR0` . Gilt *Timer Mode 0*, wird die Schleife nicht ausgeführt, da keine Zählimpulse stattfinden.

Zunächst wird innerhalb der Schleife die Haupt-Rampe gezeichnet, welche einen linear steigenden Zählerstand abbilden soll. Die Steigung ist konstant, es handelt sich also nicht um eine Veränderung der Größe bei variablem Seitenverhältnis. Grundlage dessen ist die Berechnung der Ziel-Breite, welche durch `drawnWidth + (200 - targetHeight) * ramp` gegeben ist. Der Skalierungsfaktor `ramp` ist standardmäßig `1` , wird jedoch zu späterem Zeitpunkt andere Werte annehmen, je nach gewähltem Eingangsteiler oder eingestellter Zoomstufe. Der bislang gezeichnete Graph ist `Abbildung 10` zu entnehmen. Wichtig: Die Variable `targetHeight` sowie die später verwendeten `TAxCCRn_pos` -Werte enthalten jeweils den Abstand zwischen der Box-Oberkante und dem tatsächlichen Wert. Daher ist für einen Rückschluss auf die Höhe vom Ursprung eine Subtraktion notwendig. Zudem sind die 16-Bit-Eingabewerte auf 200px normiert, sodass der Wertebereich von `65536` auf `200` herunter skaliert wird. Die Berechnung lautet `TAxCCR0_pos = ((65536 - TAxCCR0_val) / 65536) * 200`.

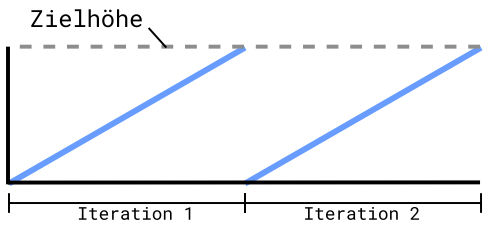


Abbildung 10: *Zeichnen der aufsteigenden Rampe im Up- und Continous-Modus*

Im Modus 3 (Up/Down) ergibt sich aufgrund der fallenden Rampe eine zusätzliche Bedingung. Innerhalb der gleichen Schleifen-Iteration wird eine zweite Linie gezeichnet, sodass die gesamte Breite eines Up/Down-Turnus der doppelten Breite der steigenden Up- und Continous-Rampe entspricht. Die entsprechende Berechnung lautet folgend `drawnWidth + 2 * (200 - targetHeight) * ramp`, wobei sich bei `(200 - targetHeight) * ramp` der Scheitelpunkt befindet. Eine entsprechende Darstellung liefert Abbildung 11. Das Erstellen des geraden Abschlusses im Up- und Continous Mode gestaltet sich als trivial, da die zu zeichnende Höhe bekannt ist.

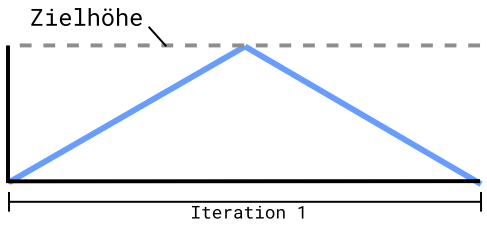


Abbildung 11: *Zeichnen der absteigenden Rampe im Up/Down-Modus; die Iteration wird aufgrund des konstanten Skalierungsfaktors doppelt so groß*

4.3.2 | Geometrische Berechnung der Schnittpunkte

Da nun die Timer-Verläufe gezeichnet sind, müssen die Schnittpunkte für die spätere Evaluation ermittelt werden. Die Schnittpunkte betreffen einerseits die Register `CCRn`, jedoch liefert auch der Timer-Verlauf selbst durch die Grenzen von `TAxR=CCR0 (=EQU0)` sowie `TAxR=0 (=TAIFG)` einige Schnitt- und Berührungspunkte. Mit der Berechnung dieser wird im Folgenden begonnen.

Abbildung 10 und Abbildung 11 ist zu entnehmen, dass die Grenzen einer jeden Iteration bereits einen Schnittpunkt mit der X-Achse kennzeichnen. Entsprechend ist durch einen Abgriff der Laufvariable `drawnWidth` ein direkter Rückschluss auf die `TAIFG`-Flags möglich. Weil die Iteration im Up/Down-Modus automatisch eine doppelte Breite annimmt, sind dahingehend keine weiteren Anpassungen nötig.

Für die Berechnung der `CCR0`-Schnittpunkte ist zunächst zu beachten, dass diese einerseits als Extrempunkt (*Peak*) im Up/Down- und Up-Mode auftreten, andererseits im Continous-Modus nur einen regulären Schnittpunkt darstellen. Entsprechend ergeben sich für `CCR0` drei notwendige Methoden der Ermittlung:

1. Im Up-Mode entspricht die X-Position von `CCR0` lediglich der Laufvariable `drawnWidth`; die Y-Position ist die Zielhöhe

2. Im Continous-Mode gilt für die Berechnung das nachfolgend erläuterte Vorgehen wie für `CCRn`
3. Im Up/Down-Modus entspricht die X-Position `Iterationsbreite/2`; die Y-Position ist die Zielhöhe

Die Berechnung von `CCRn` sowie `CCR0` im Continous-Mode ergibt sich ähnlich wie die Ermittlung der Timer-Breite. Jedoch wird als Begrenzung der Höhe nicht die Zielhöhe angegeben, sondern die jeweiligen `CCRn`-Registerwerte. Weil jeder Turnus im Grunde gleich ist, wird die `CCRn`-Breite mit `drawnWidth` als Ausgangspunkt errechnet. `CCR0` bis `CCR2` erhalten eigene Laufvariablen, wie in `Codeblock 2` gezeigt.

```
// Timer-X-Schnittpunkt
drawnWidth = drawnWidth + (200 - targetHeight) * ramp;

// CCRn-X-Schnittpunkt
drawnWidthTAxCCR1 = drawnWidth + (200 - TAxCCR1_pos) * ramp;
```

Codeblock 2: *Die Berechnung der Timer-X- und CCRn-X-Schnittpunkte: `drawnWidth` dient aus Ausgangspunkt, von welchem aus anhand der fiktiven Zielhöhe `TAxCCR1_pos` die absolute X-Position eines `CCRn`-Schnittpunkts ermittelt wird*

Die Anwendung dieser einzelnen Schnittpunkte zeigt `Abbildung 12`, wobei `CCRn` repräsentativ für alle Registerwerte steht, für welche gilt `CCRn_pos` ≠ `targetHeight`. Entsprechend ausgenommen davon ist `CCR0` im Modus 1 und 3. Im Up/Down-Modus wird die vorhandene Symmetrie ausgenutzt, da der Abstand zwischen der linken Iterationsgrenze und dem `CCRn`-Schnittpunkt betragsmäßig dem Abstand zur rechten Grenze entspricht. Durch Subtraktion des Betrags von der Gesamtbreite kann daher auf die absolute Position innerhalb des Turnus rückgeschlossen werden. Verdeutlichen soll dies `Abbildung 13`.

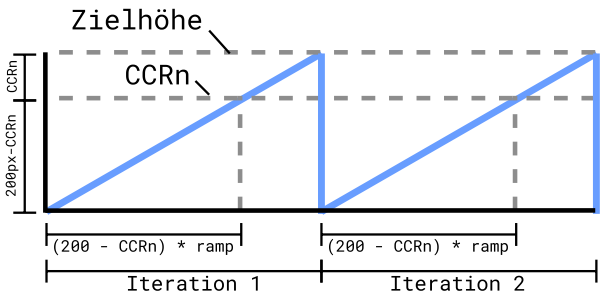


Abbildung 12: *Berechnung eines generischen `CCRn`-Schnittpunks im Up-/Continous Modus*

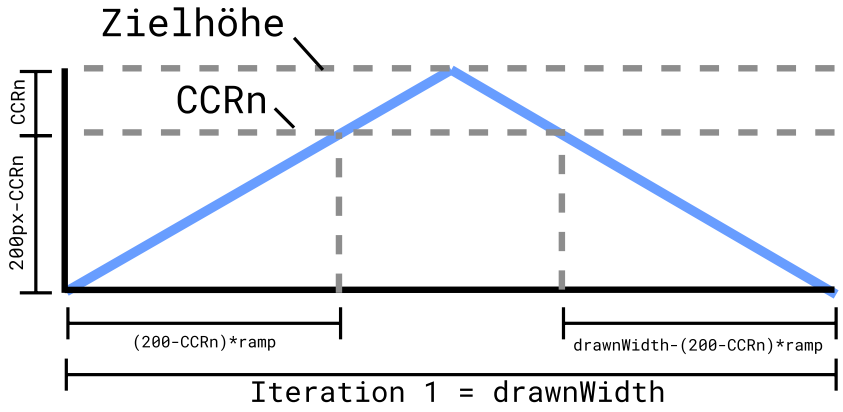


Abbildung 13: *Berechnung der generischen `CCRn`-Schnittpunke im Up/Down-Modus: Die Symmetrie vereinfacht die Berechnung*

4.3.3 | Vorteiler, Zoom und X-Achsenbeschriftung

Eine fortlaufend verwendete Variable ist der Integer `ramp`, der als Skalierungsfaktor dient. Er beschreibt den Kehrwert der Steigung der Timer-Rampe und ist im Grundzustand auf den Wert `1` festgelegt, was einer Steigung von 100 % beziehungsweise einem Breitengewinn von `1px` pro `1px` Höhe entspricht.

Der Nutzer erhält zwei Möglichkeiten der Anpassung dieses Wertes: Entweder wird ein entsprechender Vorteiler gesetzt (`Bit 7-8` im `TAxCTL`-Register) oder es wird der Schieberegler zur Zoomstufe genutzt. Beide Einstellungen bewirken einen gleichen Einfluss auf die Skalierung der Darstellung, sodass sich ein Zoom von 50 % und ein Eingangsteiler von `1/2` aufheben und in der gleichen Skalierungsstufe von `1` münden. Grafisch haben somit beide Eingänge die identische Wertigkeit.

Weil jedoch auch eine X-Achsenbeschriftung vorgesehen ist, funktioniert das zuvor genannte Vorgehen nur für Elemente, die zeitlich nicht absolut angegeben werden müssen. Im Gegensatz dazu spielt für die Achsenbeschriftung die Zeitabhängigkeit eine herausragende Rolle, sodass zwar die Positionierung nach der entsprechenden Skalierungsstufe erfolgt, die angezeigten Ziffern aber unabhängig von der Zoomstufe bleiben müssen. Dies hat den Grund, dass im vorher genannten Beispiel der korrekte zeitliche Bezug verloren gegangen ist, da ein Eingangsteiler von `1/2` die auf das Zählwerk einwirkende Taktdichte halbiert und den Timer nur halb so schnell zählen lässt. In allen zuvor genannten Berechnungen wurde mit `ramp` gerechnet – dem kumulierten Wert aus Vorteiler und Zoom. Die Schnittpunkte beziehen sich somit auf Pixel (und nicht

auf Sekunden oder Timer-Zählwerte). Daher steckt in jeder Angabe automatisch auch ein parasitärer Zoom, der die zeitliche Einordnung verfälscht.

Als Anker für ein X-Achsen-Label wird jede Schnittpunkt-Flag gewählt, da sie eine gute Orientierung für den Zählerstand des Timers bieten. Für die entsprechende Berechnung (in ms) wird zunächst der Zoom (welcher in der X-Position eines jeden Schnittpunkts enthalten ist) herausgerechnet und anschließend mit einem Umrechnungsfaktor verrechnet. Dieser Faktor beschreibt die Wertigkeit eines Pixels im Zeitbereich und beträgt `3.051758` und entsteht aus dem Verhältnis zwischen der skalierten Timer-Höhe von 200px und dem tatsächlich möglichen Timerwert 65535.

$$\frac{200}{2^{16}} = 3,051758$$

Zudem ist zur besseren Lesbarkeit eine minimale Breite für jedes Label vorgesehen. Kann diese Breite nicht eingehalten werden, werden die direkten Nachbarn ausgeblendet und erscheinen erst wieder, wenn eine höhere Zoomstufe gewählt wird.

4.3.4 | Speicherformat zur weiteren Verarbeitung

Um die Schnittpunkte entsprechend zu speichern wird das Array `graphIntersections` verwendet, das alle Schnittpunkte unsortiert aufnimmt. Durch die Funktion `array.push({})` werden jedem Schnittpunkt entsprechende Werte beigefügt. Das Erstellen dieser Elemente ist in Codeblock 3 dargestellt. Je nachdem, in welcher Timer-Umgebung der Schnittpunkt entsteht, werden über die Parameter die Flankentypen gespeichert. Dies dient einerseits der lehrhaften Nachvollziehbarkeit für den Nutzer in späteren Abbildungen, hat jedoch auch einen Einfluss auf die Erzeugung des Ausgangssignals.

Nach Abschluss dieses Speicherns ist das Zeichnen des Timers und die Ermittlung der Interrupts abgeschlossen. Alle nachfolgenden Programmteile arbeiten mit dem erzeugten `graphIntersections`-Array weiter.

```
// Für einen Schnittpunkt CCR1 in allen Modi
graphIntersections.push({
    "x": drawnWidth +
         (200 - TAxCCR1_pos) * ramp,
    "y": TAxCCR1_pos,
    "limitType": "TAxCCR1",
    "edgeType": "rise"
});

// für einen symmetrischen Schnittpunkt
// CCR1 im Up/down-Modus
graphIntersections.push({
    "x": drawnWidth* -
         (200 - TAxCCR1_pos) * ramp,
    "y": TAxCCR1_pos,
    "limitType": "TAxCCR1",
    "edgeType": "fall"
});
```

```
// Unterscheidung: CCR0 kann auch als
// Peak auftreten
graphIntersections.push({
    "x": drawnWidthTAxCCR0,
    "y": TAxCCR0_pos,
    "limitType": "TAxCCR0",
    "edgeType": "peak"
});

// Für einen Schnittpunkt CCR0 im
// Continous Mode
graphIntersections.push({
    "x": drawnWidthTAxCCR0,
    "y": TAxCCR0_pos,
    "limitType": "TAxCCR0",
    "edgeType": "rise"
});
```

Codeblock 3: *Speichern der Schnittpunkte auf Grundlage des Timer-Modus und dem Registernamen (Hinweis:* `drawnWidth*` *repräsentiert hier die in der darauffolgenden Iteration gezeichnete Breite, also* `drawnWidth* = drawnWidth + Iterationsbreite`)

4.4 Anzeige der Interrupts

Nach Abbildung 8 folgt nach der Erzeugung der Schnittpunkte die Anzeige der Interrupts. Die Schnittpunkte werden in diesem Zwischenschritt als Projektion des Timers dargestellt. Die anzuzeigenden Interrupts lauten einerseits `TAIFG`, andererseits sollen auch `EQU0-EQU2` aufgeführt sein. Die Anzeige listet die einzelnen Schnittpunkte gruppiert auf, sodass für `TAIFG` die Schnittpunkte auch nach `TAIFG` gefiltert werden. Zusätzlich wird beim entsprechenden Interrupt der Flankentyp des Zählers gezeigt, sodass `TAIFG` immer einem Tal (»Through«) und `EQU0` im Up/Down- und Up-Mode einem Höhepunkt (»Peak«) entspricht. Codeblock 4 zeigt einen beispielhaften Inhalt von `graphIntersections = []`, Abbildung 14 die zugehörige Grafik. Der Y-Wert wird dazu genutzt die gestrichelte, vertikale Linie am oberen Ansatz auf der korrekten Höhe zu zeichnen. Das gesamte Array wird iterativ abgearbeitet.

```
[
    [x: 0,   y: 0,   edgeType: "through", limitType: "TAIFG"],
    [x: 100, y: 50,  edgeType: "rise",    limitType: "TAxCCR2"],
    [x: 200, y: 125, edgeType: "rise",    limitType: "TAxCCR1"],
    [x: 300, y: 175, edgeType: "peak",    limitType: "TAxCCR0"],
    [x: 300, y: 0,   edgeType: "through", limitType: "TAIFG"],
    ...
]
```

Codeblock 4: *Beispielhafter Inhalt des Arrays* `graphIntersections`*, welches die Schnittpunkte für die Ausgänge 0-2 und TAIFG beinhaltet.*

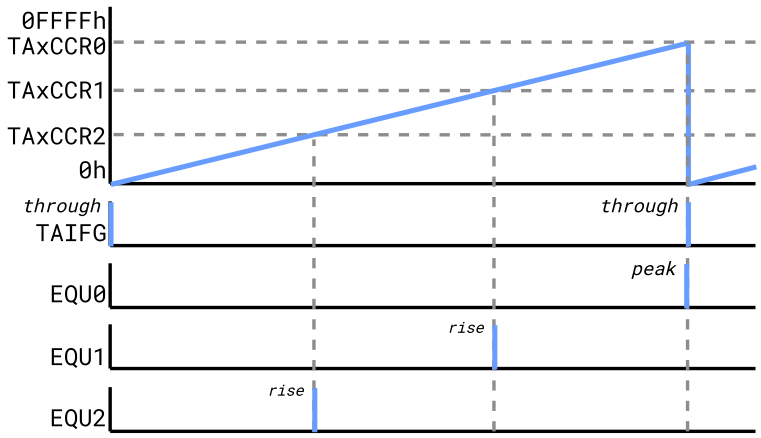


Abbildung 14: *Projektion der Schnittpunkte* `graphIntersections` *als Interrupts*

4.5 Erzeugen des Ausgangssignals

Aus den Schnittpunkten wird nun anhand der von TI definierten Output-Modi das Ausgangssignal entwickelt. Dazu wird ein entsprechendes Pattern verwendet, das die Pegelwechsel und dessen Bedingungen entsprechend gespeichert hat. Anschließend wird die bereits im Theorieteil erwähnte Unstimmigkeit in der Herstellerdokumentation durch Messungen untersucht und das korrekte Pattern ermittelt. Zuletzt erfolgt die Ausgabe der finalen Signale.

4.5.1 | Mapping der Ausgangsmodi – Output-Pattern

Die Ausgangsmodi sind als Array gespeichert, welches die Pegelwechsel gespeichert hat. Dabei spielen verschiedene Bedingungen eine Rolle:

- Im Up-/Continous-Mode ergibt sich laut der grafischen Erklärung von TI ein anderes Verhalten als im Up/Down-Modus. Dies muss entsprechend berücksichtigt werden. (vgl. Abbildung 5)
- Es muss unterschieden werden, ob es sich um einen `CCR0` -Schnittpunkt oder ein `CCRn` -Event handelt. Nur so kann die textuelle Definition von TI umgesetzt werden. (vgl. Tabelle 1)
- Behält die grafische Darstellung des Up/down-Mode Recht, sind einige Pegelwechsel abhängig von der Timer-Flanke zum Zeitpunkt des Schnittpunkts. Daher spielt hier auch das im Array `graphIntersections` gespeicherte `edgeType` -Attribut eine Rolle.

Zunächst wird zwischen den verschiedenen Output-Modi unterschieden. Das Pattern ist ein verschachteltes, indiziertes Array, dessen oberste Ebene durch numerische Schlüssel (0, 1, …) adressiert wird *(1)*. Der Schlüssel steht dabei für den entsprechenden Output-Modus. Jedes dieser numerischen Elemente enthält zwei benannte Modi: `default` und `up-down` *(2)*. Dies hat den Ursprung darin, dass beim direkten Vergleich zwischen Up- und Continouos-Modus und Up/Down widersprüchliche Pegelwechsel im Pattern erfasst werden müssen. Exemplarisch sei OM2 genannt, bei welchem `CCR0` einerseits ein Rücksetzen auslöst, andererseits im Up/down-Modus offenbar keine Eigenschaft hat. (vgl. Abbildung 15)

Innerhalb jedes Modus befinden sich Einträge für verschiedene Timer-Register, speziell die Unterscheidung von `CCR0` - und `CCRn` -Schnittpunkt *(3)*. Für jedes Register sind wiederum drei Signalphasen definiert, welche die Flanke berücksichtigen, sodass auch das symmetrische Up/Down-Signal erzeugt werden kann. Diese Phasen enthalten jeweils ein Array von booleschen Werten, das beschreibt, welche Ausgabezustände zu welchen Zeitpunkten aktiv sind *(4)*. Dabei enthält der erste Eintrag stets den Zielzustand, wenn der Pegel vorher `LOW` ist. Der zweite Eintrag entsprechend umgekehrt, wenn der Pegel vorher `HIGH` ist. Ein Setzen-Befehl wird somit durch `[true, true]` festgelegt, ein Reset-Befehl durch `[false, false]` und ein Toggle mit `[true, false]`. Soll keine Änderung am Signal vorgenommen werden, kann dies durch `[false, true]` realisiert werden. Das in Codeblock 5 gezeigte Verhalten wird im Pattern mit den in Abbildung 15 gezeigten Angaben erreicht.

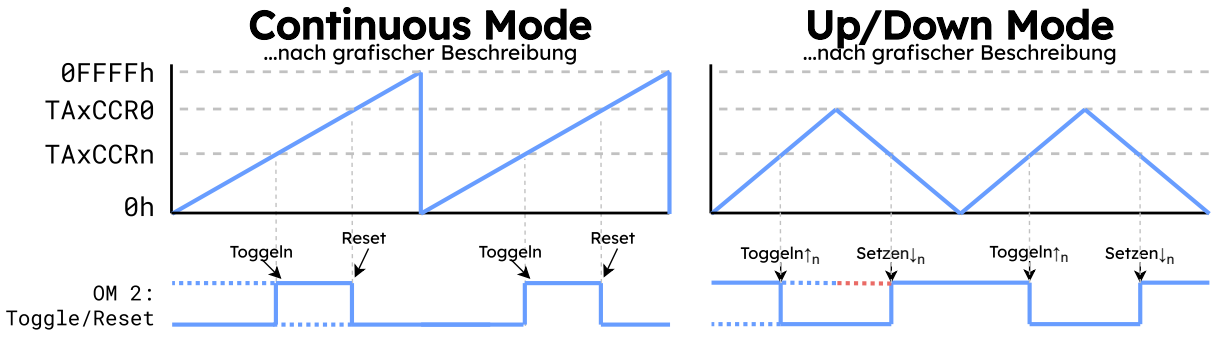


Abbildung 15: *Widersprüchlichkeiten des dokumentierten Up/Down-Mode und dem Continuous-Mode der MSP430x5xx- und MSP430x6xx-Modelle im Output Mode 2 (`toggle/Reset`); interpretierter Ausschnitt von* Abbildung 5 .

4.5.2 | Ermitteln des korrekten Ausgangsverhaltens

Aufgrund der widersprüchlichen Dokumentation des Up/Down-Modus ist es notwendig, den tatsächlichen Ausgang zu untersuchen, den die Hardware des MSP430 erzeugt.

```
outputModePatterns = {
    ...,
    2:  // <1>
    {
        "default": {  // <2>
            "TAxCCR0": {
                "rise": [false, false],   // <3> <4>
                "peak": [false, false],
                "fall": [false, false]
            },
            "TAxCCRn": {
                "rise": [true, false],
                "peak": [true, false],
                "fall": [true, false]
            }
        },
        "up-down": {  // <2>
            "TAxCCR0": {
                "rise": [false, true],
                "peak": [false, true],
                "fall": [false, true]
            },
            "TAxCCRn": {
                "rise": [true, false],
                "peak": [false, true],
                "fall": [true, true]
            }
        }
    },
    3:
        {...},
    ...
}
```

Codeblock 5: *Ausschnitt aus dem Output-Pattern: Output Mode 2 (`toggle/Reset`)*

Für die nachfolgenden Messungen wird ein Mikrcontroller der Familie *MSP430G2xx* verwendet – konkret handelt es sich um das Launchpad-Kit *MSP-EXP430G2ET* von TI. Damit die Messungen des *MSP430G2xx* auch auf die *MSP430x5xx/x6xx*-Familie übertragbar sind, muss sichergestellt sein, dass sich die Peripherie nicht unterscheidet. Abbildung 16 zeigt eine direkte Nebeneinanderstellung der Blockschaltbilder der *MSP430x5xx/x6xx*-Familie und der MSP430F2xx/MSP430G2xx-Familie. Dabei wird ersichtlich, dass sich der Aufbau bis auf die unterschiedliche Anzahl von `CCR`-Einheiten, den fehlenden `TAIDEX`-Vorteiler und die verschieden ausgeführte Darstellung der Busbreite nicht unterscheidet. Die Validität der durchgeführten Hardware-Tests auf dem MSP430G2xx wird durch die Herstellerdokumentation gestützt: In der Änderungsdokumentation zur Timer_A-Architektur der *x5xx*-Familie wird explizit bestätigt, dass die Funktionalität »*fully compatible with previous Timer_A offerings on the MSP430F2xx […] families*« ist. [11, S. 5]

Zudem weisen die textuellen Dokumentationen der *MSP430x5xx/x6xx*-Familie und der *MSP430F2xx/G2xx*-Familie die exakt gleichen Pegelwechsel auf (wie in Tabelle 1). [7, S. 469] [12, S. 381ff] Da somit das `Timer_A`-Peripheriemodul in der MSP430-Architektur auf derselben logischen Struktur basiert und die Register-Definitionen

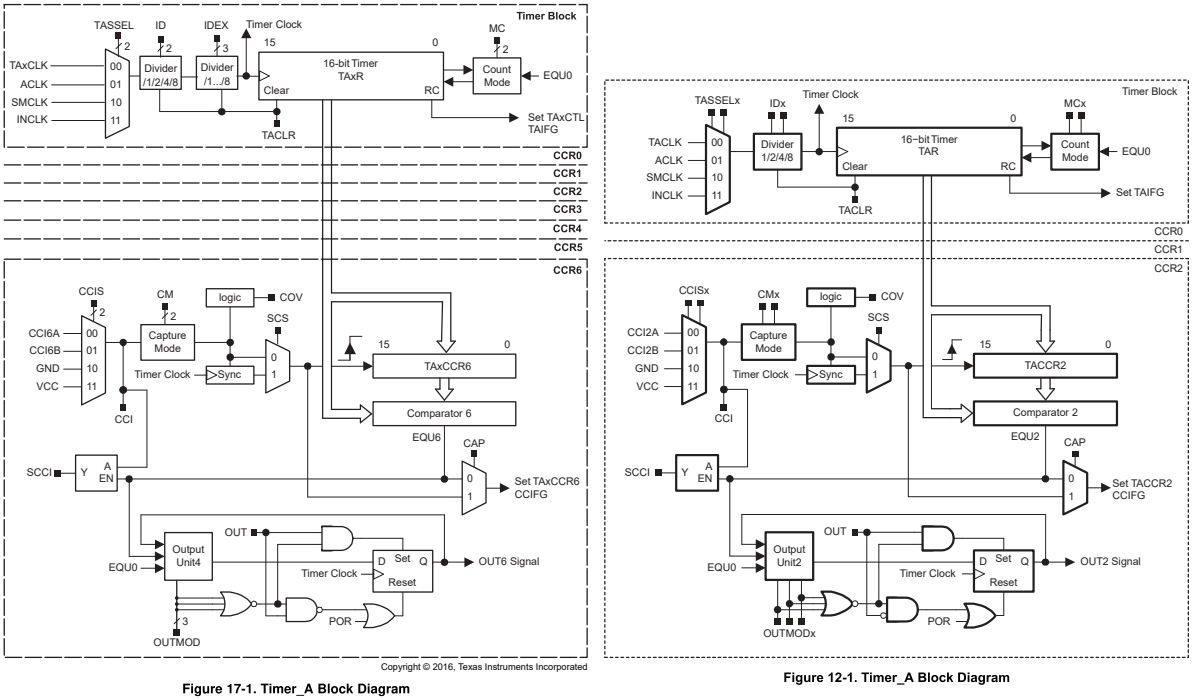


Figure 17-1. Timer_A Block Diagram

Figure 12-1. Timer_A Block Diagram

Abbildung 16: *Vergleich zwischen dem MSP430F2xx/G2xx (rechts) [12, S. 375] und dem MSP430x5xx/x6xx (links) [7, S. 462]*

textuell deckungsgleich sind, wird der *MSP430G2xx* als valides Referenzmodell für das Verhalten der Output-Unit herangezogen.

Jedoch gibt es beim direkten Vergleich der grafischen Darstellung des Up/Down-Mode Unterschiede in der Dokumentation. Im Falle der *MSP430F2xx/G2xx*-Familie deckt sich diese exakt mit der auf Basis der textuellen Beschreibung für diese Arbeit erstellten Abbildung 6 (links). [12, S. 384] Die Grafik der *MSP430x5xx/x6xx*-Familie ist ebenfalls durch Abbildung 6 (rechts) bereits bekannt. Eine direkte Gegenüberstellung bietet Abbildung 17. Etwaige grafische Diskrepanzen im Handbuch der *x5xx*-Familie stehen im Widerspruch zur eigenen textuellen Definition und werden daher als Dokumentationsfehler klassifiziert, während das Verhalten der *G2xx*-Hardware als Referenz für die korrekte Umsetzung der spezifizierten Logik dient. Im Folgenden wird das Verhalten nach Tabelle 1 als korrekt verfolgt.

Die Messungen mit dem Oszilloskop sollen die Pegelwechsel messen. Dazu wird ein C-Programm verwendet, mit welchem alle Output-Modi nacheinander an einem Pin des Mikrocontrollers ausgegeben werden können. Zur zeitlichen Einordnung wird zudem ein Interruptvektor erstellt, der einen kurzen Marker-Impuls erzeugt, wenn `TAR = CCR0` gilt. Damit auch die gestrichelten Linien simuliert werden können, darf der Pegel in einigen Modi nicht jenem im eingeschwungenen Zustand entsprechen. Daher wird

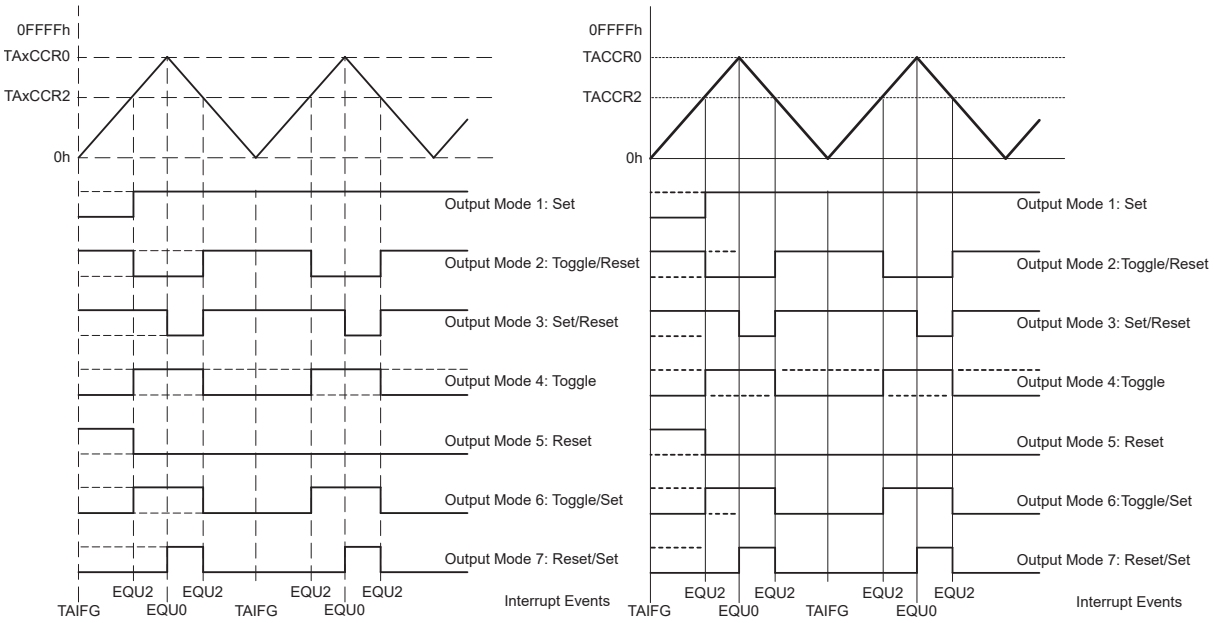


Figure 17-14. Output Example – Timer in Up/Down Mode

Figure 12-14. Output Example—Timer in Up/Down Mode

Abbildung 17: *Gegenüberstellung des spezifizierten, grafisch dargestellten Output-Verhaltens des MSP430x5xx/x6xx (links)* [7, S. 462] *und dem MSP430F2xx/G2xx (rechts)* [12, S. 384]

`TA0CCTL1 = OUTMOD_0` gesetzt, wobei für eine positive Flanke zusätzlich das `OUT`-Bit maskiert wird. Zur Auswertung wird der Ausgang der Compare-Unit 1 `TA0CCTL1` genutzt; `CCR1` ist auf den halben Wert von `CCR0` eingestellt. Abbildung 18 zeigt den Versuchsaufbau. Der original auf den MSP430 geladene Code kann Anhang 1 entnommen werden. Das Erstellen des Codes verlief KI-gestützt.

Die Aufzeichnungen zeigen, dass die textuelle Definition das Verhalten des `Timer_A` im *MSP430G2xx* (und damit auch der *MSP430x5xx/x6xx*-Familie) korrekt beschreibt. Dies gilt für alle Output-Modi. Abbildung 19 zeigt exemplarisch die direkte Nebeneinanderstellung des Output Mode 2 (Set/Reset). Dies entspricht der Darstellung in der Dokumentation des *MSP430G2xx* (Abbildung 17 , rechts) und der auf Basis der textuellen Erklärung erstellten Grafik (Abbildung 6 , links). Im Anhang 2 sind alle Aufnahmen der in Abbildung 6 mit Differenzen markierten Output-Modes aufgelistet.

4.5.3 | Besonderheiten von CCR0

TI schreibt in der Dokumentation: »*Output modes 2, 3, 6, and 7 […] are not useful for output unit 0 because EQUn = EQU0.*« [7, S. 469] Dies wird nicht näher erläutert, wenngleich das Treffen dieser Einstellung (beziehungsweise das Auslesen von `OUT0`) technisch problemlos möglich ist. Weil dies der Fall ist, muss das korrekte Verhalten auch für den Simulator durch Messen ermittelt werden. Dazu wird der gleiche Aufbau wie in

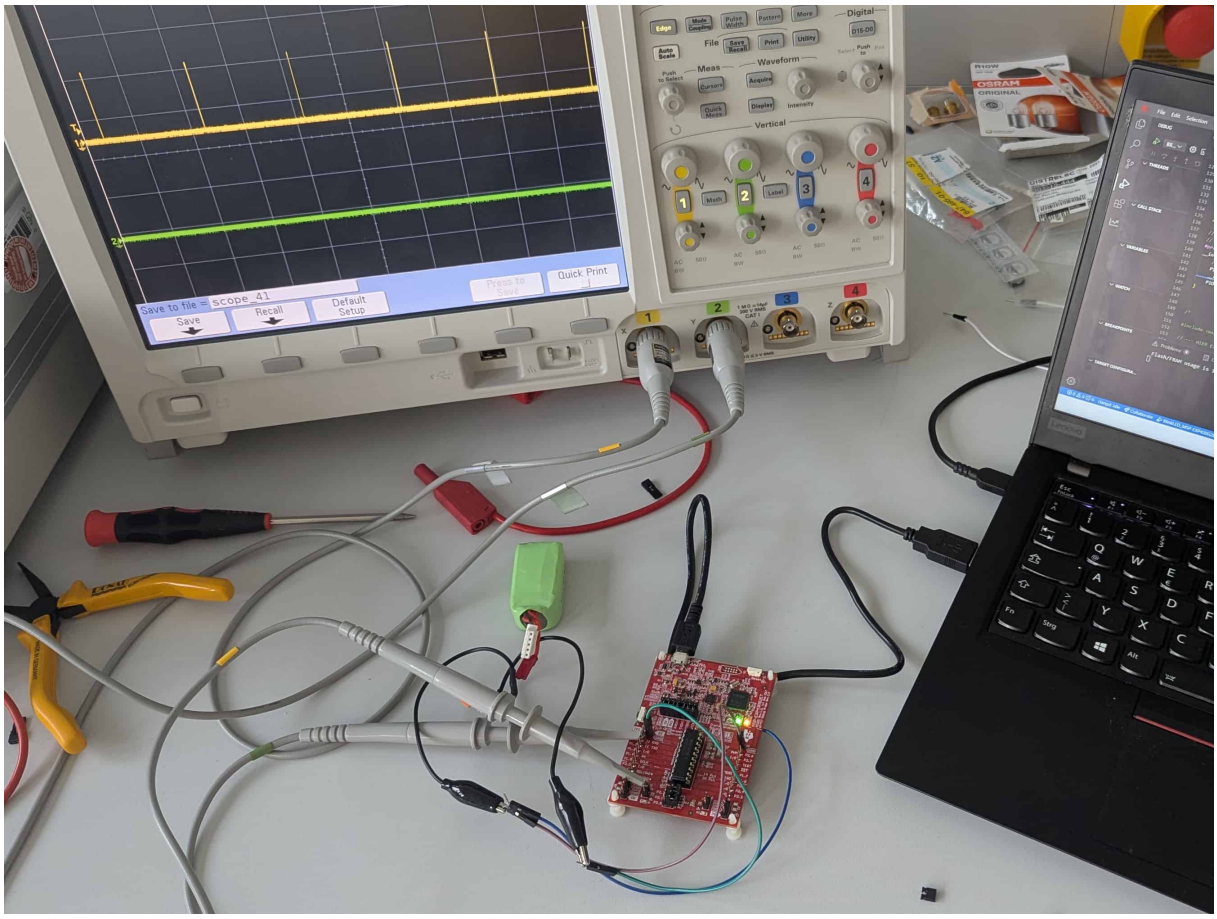


Abbildung 18: *Versuchsaufbau für das Vermessen des MSP-EXP430G2ET: Der gelbe Tastkopf (`TAIFG`-Identifikation) ist an Pin `1.0` angeschlossen; der grüne Tastkopf (Ausgangssignal `TA0CCTL1`) liegt an Pin `1.6`. Zur Programmierung wird die Entwicklungsumgebung CCSTUDIO von TI genutzt.*

Kapitel 4.5.2 genutzt, jedoch ohne das Festlegen des Startwertes. Der grüne Tastkopf liegt nun an Pin `1.1` (TA0.0) mit dem Ausgangssignal von `TA0CCTL0`. Die Messwerte sind Abbildung 20 zu entnehmen.

Ein konkretes Muster ist dabei nicht zu erkennen. Zwar scheint der letzte Befehl (`~/Reset`, `~/Set`) zu unterwiegen, jedoch wird auch ein `Toggle/~`-Befehl nicht vollständig ausgeführt. Für die Entwicklung des Demonstrators lässt sich aber vereinfacht ableiten, dass im OM2 und OM3 der `SET`-Befehl überwiegt, während es bei OM6 und OM7 der `RESET`-Befehl ist. Dies muss im Output-Pattern selbst sowie der Anwendung des Output-Pattern berücksichtigt werden.

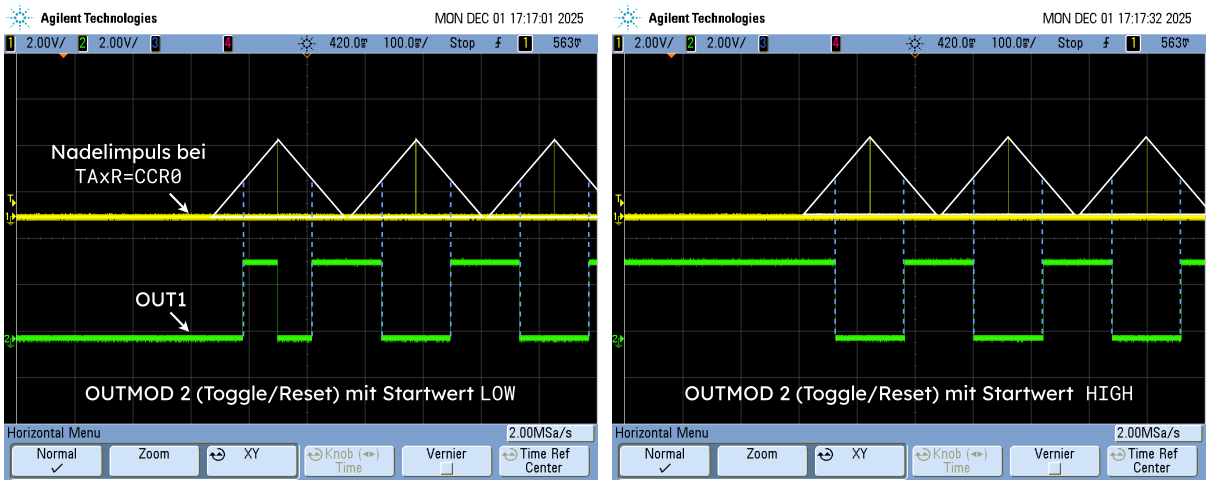


Abbildung 19: *Messen des `OUT1`-Signals bei `OUTMOD2` mit dem Oszilloskop: Die Aufnahmen zeigen einen `Reset`-Impuls bei `CCR0=TAR` und einen `Toggle`-Impuls bei `CCR1=TAR/2`. Die Schenkel des Timer-Dreiecks wurden zur besseren Anschaulichkeit im Nachhinein erstellt, ebenso die Projektion der Schnittpunkte auf den Ausgang. Die restlichen Messungen sind* Anhang 2 *zu entnehmen.*

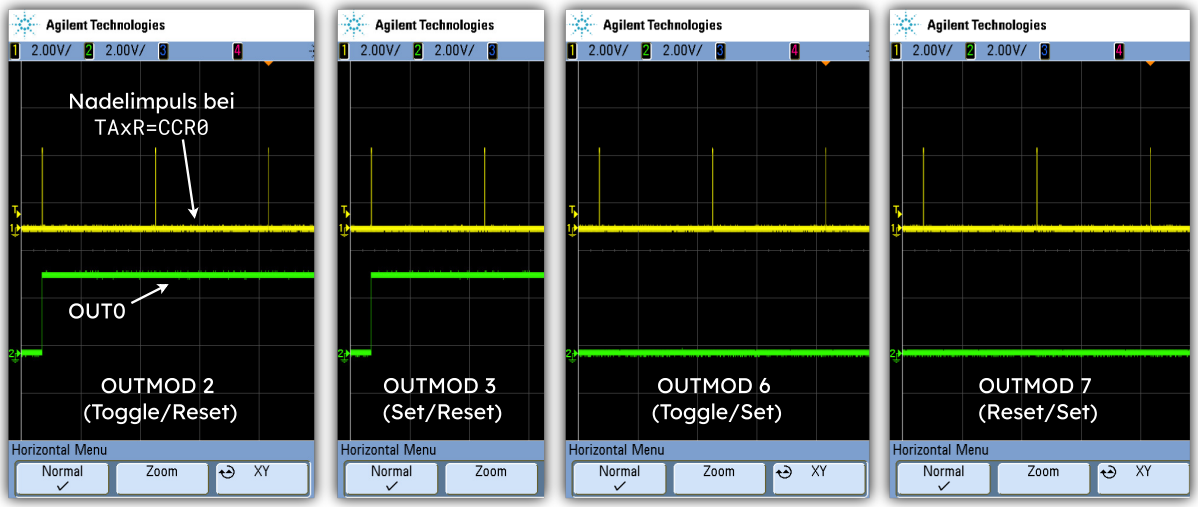


Abbildung 20: *Das mit dem Oszilloskop gemessene `OUT0`-Signal in den Output Modi 2, 3, 6 und 7 (v.l.n.r.)*

4.5.4 | Nutzen des Output-Pattern

Mit dem in Kapitel 4.5.2 und Kapitel 4.5.3 ermittelten Output-Verhalten muss nun ein entsprechendes Programm entwickelt werden, welches das Output-Pattern korrekt anwendet. Dem Programmcode liegen zwei Arrays bei: Eines, das das grafische Verhalten nachbildet, und eines, das sich strikt an die textuelle Definition nach Tabelle 1 hält. Für die Implementierung steht es daher frei, welches Pattern verwendet werden soll. Weil

sich in den Versuchen die Grafiken als falsch herausgestellt haben, wird im Folgenden das Pattern für die textuelle Definition genutzt. Weil dennoch die Auswertelogik für die Flankentypen enthalten ist, entsteht ein entsprechender Overhead, beziehungsweise existieren identische Definitionen für `default` und `up-down`.

Um das Verhalten aus Kapitel 4.5.3 abbilden zu können ist zudem die weitere Information notwendig, wie die Simulation handeln soll, wenn der »bistabile« Fall von `CCRn=CCR0` eintritt. Dazu wird ein weiterer Flankentyp eingeführt, welcher ein definiertes Ausgangsverhalten festlegt, wenn `CCR0` als `CCRn` aufgerufen wird. Für OM2 und OM3 lautet es `"bistable": true` und für OM6 und OM7 `"bistable": false`. Der Wert wird jeweils beim ersten Erreichen von `CCR0 = TAR` gesetzt.

Das Programm arbeitet iterativ alle Schnittpunkte aus `graphIntersections[]` ab, welche nach X-Koordinate sortiert sind. Weil für jedes `OUTn` eine eigene Ausgabe erzeugt wird, können alle Compare-Einheiten parallel abgearbeitet werden. Dazu wird zunächst die Funktion des Schnittpunkts ermittelt, indem `CCR1` und `CCR2` die allgemeine Wertigkeit von `CCRn` zugeordnet wird. `CCR0` erhält dessen Bedeutung, wenngleich es durch das Attribut `"bistable": ...` automatisch zum vordefinierten Wert gezwungen wird, unabhängig vom Schnittpunkt-Typ oder der Flanke. Der Fall `CCR0` wird durch Codeblock 6 verdeutlicht; Codeblock 7 zeigt exemplarisch den Ablauf für das Ermitteln des `OUT1`-Signals, das nach den Schnittpunkten `CCR0` und `CCR1` filtert. Die Funktion `useOutputModePattern()` wird darauffolgend beschrieben.

```
if (originalLimitType === "TAxCCR0") {
    element["limitType"] = "TAxCCRn";
    useOutputModePattern(0, element, outputMode0, timerMode, true);
}
```

Codeblock 6: *Aufruf von `CCR0` als `CCRn` beim Anwenden des Output-Pattern: Das Signal `OUT0` ist ausschließlich definiert durch `CCR0`. Durch die zuletzt gesetzte Flag wird erlaubt, dass der bistabile Fall eintreten darf und der Anwendung eines entsprechend im Output-Pattern vordefinierten Wertes zugestimmt wird.*

`useOutputModePattern()` wird bei jedem Schnittpunkt aufgerufen. Inhalt der Funktion ist grundsätzlich das korrekte Entnehmen der Pegelwerte aus dem Pattern, abhängig vom vorherigen Wert, dem Output-Modus, dem Timer-Typ und dem Schnittpunkt-Typ. Auf Basis der X-Koordinate in `graphIntersections[]` wird jeder Pegel mit dem entsprechenden X-Wert (von/nach) versehen. Auch wenn kein Pegelwechsel stattfindet, wird ein neues Array-Element erzeugt. Die korrekte Darstellung obliegt einem späteren Programmabschnitt.

```
if (["TAxCCR0", "TAxCCR1"].includes(originalLimitType)) {
    if (originalLimitType === "TAxCCR1") {
        element["limitType"] = "TAxCCRn";
    } else {
        element["limitType"] = originalLimitType; // TAxCCR0
    }
    useOutputModePattern(1, element, outputMode1, timerMode);
}
```

Codeblock 7: *Aufruf von `CCR0` und `CCRn` beim Anwenden des Output-Pattern: Durch die Kombination entsteht das Signal `OUT1`. Im Gegensatz zu `OUT0` ist hier kein Konflikt zu erwarten, daher wird `CCR1` als `CCRn` betrachtet (`originalLimitType`) und `CCR0` als `CCR0`.*

Alle Daten werden in das Array `outputLevels[outputIndex]` geschrieben, wobei `outputIndex` für den jeweiligen OUT-Kanal steht. Einen beispielhaften Inhalt dieses Arrays zeigt Codeblock 8.

```
[[
    {x_from: 0,       x_to: 350,  level: false},
    {x_from: 350,     x_to: 375, level: true},
    {x_from: 375,     x_to: 750,  level: true},
  ],[
    {x_from: 0,       x_to: 350,  level: false},
    {x_from: 350,     x_to: 375, level: true},
    {x_from: 375,     x_to: 750,  level: true},
  ],[
    {x_from: 0,       x_to: 350,  level: false},
    {x_from: 350,     x_to: 375, level: true},
    {x_from: 375,     x_to: 750,  level: true},
]]
```

Codeblock 8: *Beispielhafter Inhalt des Arrays `outputLevels[]`, welches das generierte Ausgangssignal für die Ausgänge 0-2 beinhaltet.*

4.5.5 | Ausgabe des Output-Pegels

Als letzten Schritt der Darstellung muss der in `outputLevels[]` gespeicherte Pulsverlauf dargestellt werden. Dazu wird – ähnlich wie in Kapitel 4.4 – das Array abgearbeitet und auf dem entsprechenden `OUTn`-Kanal angezeigt. Der wesentliche Unterschied liegt in der durchgängigen Verbindung der einzelnen X-Koordinaten miteinander sowie der entsprechenden Anzeige der Pegelwechsel. Grundsätzlich wird der Pegel eines Segments mit dem vorherigen Segment verglichen, sodass gerade Abschlüsse gezeichnet werden. Dazu wird auf dem Canvas jeweils die Ausgangsposition als Startpunkt für das Zeichnen einer Linie gewählt und bis zum Endpunkt des Segments gezogen. Abbildung 21 zeigt eine reale Bildschirmaufnahme aus dem Programm mit allen erörterten Schritten der vorherigen Kapitel.

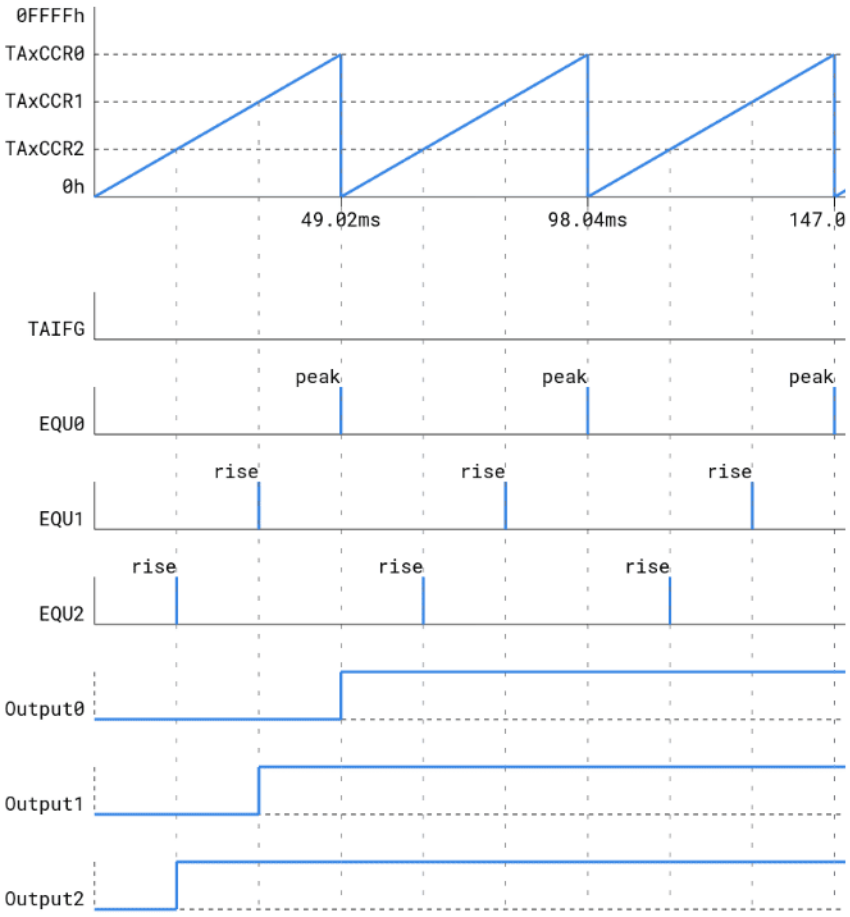


Abbildung 21: *Abschließende Bildschirmaufnahme des Timers, der Projektion der Schnittpunkte und der Darstellung des Ausgangssignals*

4.6 Kontextsensitive Erläuterungen

Aus didaktischen Gründen soll das Programm eine Erklärung beifügen, wie die Nutzereingaben interpretiert und aktuell dargestellt werden. Den entsprechenden Registern `TAxCTL` und `TAxCCTLn` ist eine blaue Hinweis-Box zugeordnet.

Für `TAxCTL` wird der Timer-Modus, die Taktquelle, der Eingangsteiler und das aktive `TAIE`-Bit angezeigt. Durch entsprechendes Mapping werden die aktiven Bits in für Menschen lesbare Einstellungen übersetzt. Seitens `TAxCCTLn` wird der Output-Modus und das Out-Bit angezeigt. Fehlerhafte Eingaben werden in einer separaten Box (wie in `Kapitel 4.2` erläutert) angezeigt. Eine Darstellung der Erklärungen bietet `Abbildung 22`.

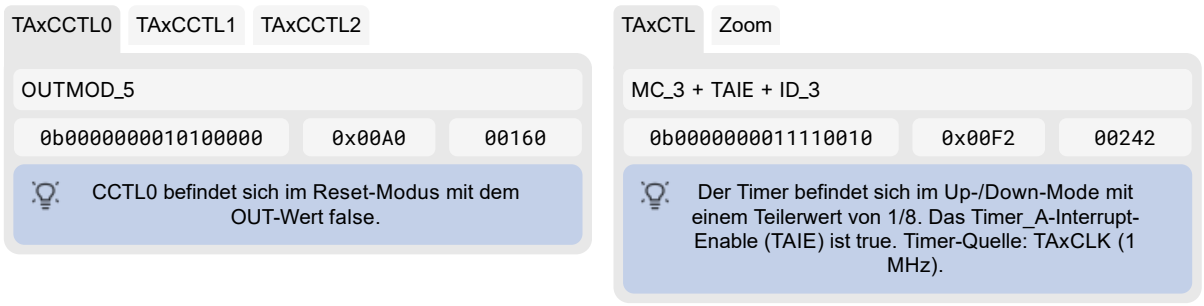


Abbildung 22: *Darstellung der Erklärungen unterhalb der Eingabemaske von* `TAxCTL` *und* `TAxCCTLn`

5 Zusammenfassung

In dieser Arbeit wurde ein HTML/JavaScript-Demonstrator für die Compare-Einheit des MSP430-Mikrocontrollers von Texas Instruments entwickelt. Ziel war es, Studierenden eine praxisnahe Möglichkeit zu bieten, das Verhalten der Timer-Einheit zu erforschen und eigene Ausgangskonfigurationen auszuprobieren.

Der Demonstrator bildet die in der offiziellen Dokumentation beschriebenen Grafiken interaktiv nach und liefert ergänzende didaktische Hinweise, um den Einstieg in die hardwarenahe Programmierung zu erleichtern und Eingabefehler zu vermeiden.

Während der Entwicklung zeigte sich, dass die offizielle Dokumentation teilweise widersprüchliche Angaben enthielt. Durch gezielte Messungen mit einem Oszilloskop konnten diese Unstimmigkeiten identifiziert und korrigiert werden. Dadurch konnte ein realistischer und funktionaler Demonstrator erstellt werden, der sowohl für Lehr- als auch für Lernzwecke eingesetzt werden kann.

6 Ausblick

Obwohl alle ursprünglich definierten Ziele dieser Arbeit erreicht wurden, lassen sich aus der entwickelten Simulation einige Erweiterungen ableiten. Diese bieten die Möglichkeit, den Demonstrator noch praxisnäher zu gestalten.

- Die Einbindung von Interrupts (etwa über einen Button-Klick) würde die *Darstellung der Capture-Einheit* ermöglichen. Hierfür müsste der Kern der Simulation auf eine zeitabhängige Struktur umgestellt werden, um Zeitmessungen realistisch abzubilden.
- Zusätzlich zum Zeitdiagramm könnte eine simulierte, blinkende LED dargestellt werden, um das *PWM-Signal anschaulich zu visualisieren*. Dies stellt insbesondere bei Monitoren eine technische Herausforderung dar, um ein entsprechendes Dimmen korrekt zu demonstrieren.
- Die Simulation könnte um zusätzliche Ausgänge (über CCR2 hinaus) erweitert werden, um *komplexere Timer-Szenarien* zu ermöglichen.
- Eine *interaktive Einführung* könnte Studierende gezielt durch die Funktionen führen, relevante Register vorstellen und einige sinnvolle Konfigurationen vorauswählen und erklären.

7 Quellenverzeichnis

[1] Matthias Sturm. *Mikrocontrollertechnik: Am Beispiel der MSP430-Familie*. 2., neu bearbeitete Auflage. Hanser eLibrary. München: Hanser Verlag, 2014. 312 S. ISBN: 978-3-446-42964-2. DOI: 10.3139/9783446429642.

[2] Manuel Jiménez, Rogelio Palomera und Isidoro Couvertier. *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. ISBN: 978-1-4614-3143-5.

[3] Texas Instruments Incorporated. *MSP430 Microcontrollers*. URL: https://www.ti.com/de-de/product-category/microcontrollers-processors/mcus/msp430/products.html (besucht am 24. 11. 2025).

[4] Winfried Gehrke und Marco Winzker. *Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller*. 8. Aufl. 2022. Berlin, Heidelberg: Springer Berlin Heidelberg, 2023. 1 S. ISBN: 978-3-662-63954-2. DOI: 10.1007/978-3-662-63954-2.

[5] Theo Ungerer. *Mikrocontroller und Mikroprozessoren*. 3. Aufl. 2010. eXamen.press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. 1 S. ISBN: 978-3-642-05398-6. DOI: 10.1007/978-3-642-05398-6.

[6] Reiner Kriesten. *Embedded Programming: Basiswissen und Anwendungsbeispiele der Infineon XC800-Familie*. Berlin/München/Boston: Walter de Gruyter GmbH, 2012. 1 S. ISBN: 978-3-486-71985-7.

[7] Texas Instruments Incorporated. *MSP430x5xx and MSP430x6xx Family - User's Guide*. März 2018. URL: https://www.ti.com/lit/pdf/slau208 (besucht am 17. 11. 2025).

[8] Texas Instruments Incorporated. *MSP430-GCC-OPENSOURCE — GCC - Open Source Compiler for MSP Microcontrollers: Header and Support Files*. URL: https://dr-download.ti.com/software-development/ide-configuration-compiler-or-debugger/MD-LlCjWuAbzH/9.3.1.2/msp430-gcc-support-files-1.212.zip (besucht am 03. 12. 2025).

[9] Peter Bühler, Patrick Schlaich und Dominik Sinner. *HTML und CSS: Semantik - Design - Responsive Layouts*. 2. Aufl. 2023. Bibliothek der Mediengestaltung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2023. 1 S. ISBN: 978-3-662-66663-0. DOI: 10.1007/978-3-662-66663-0.

[10] Mark Simon. *JavaScript for Web Developers: Understanding the Basics*. 1st ed. Berkeley, CA: Apress L. P, 2023. 1 S. ISBN: 978-1-4842-9774-2.

[11] Texas Instruments Incorporated. *MSP430F5xx Overview and Comparison to MSP430F2xx and MSP430F4xx*. Sep. 2018. URL: https://www.ti.com/de/lit/pdf/slaa396 (besucht am 20. 12. 2025).

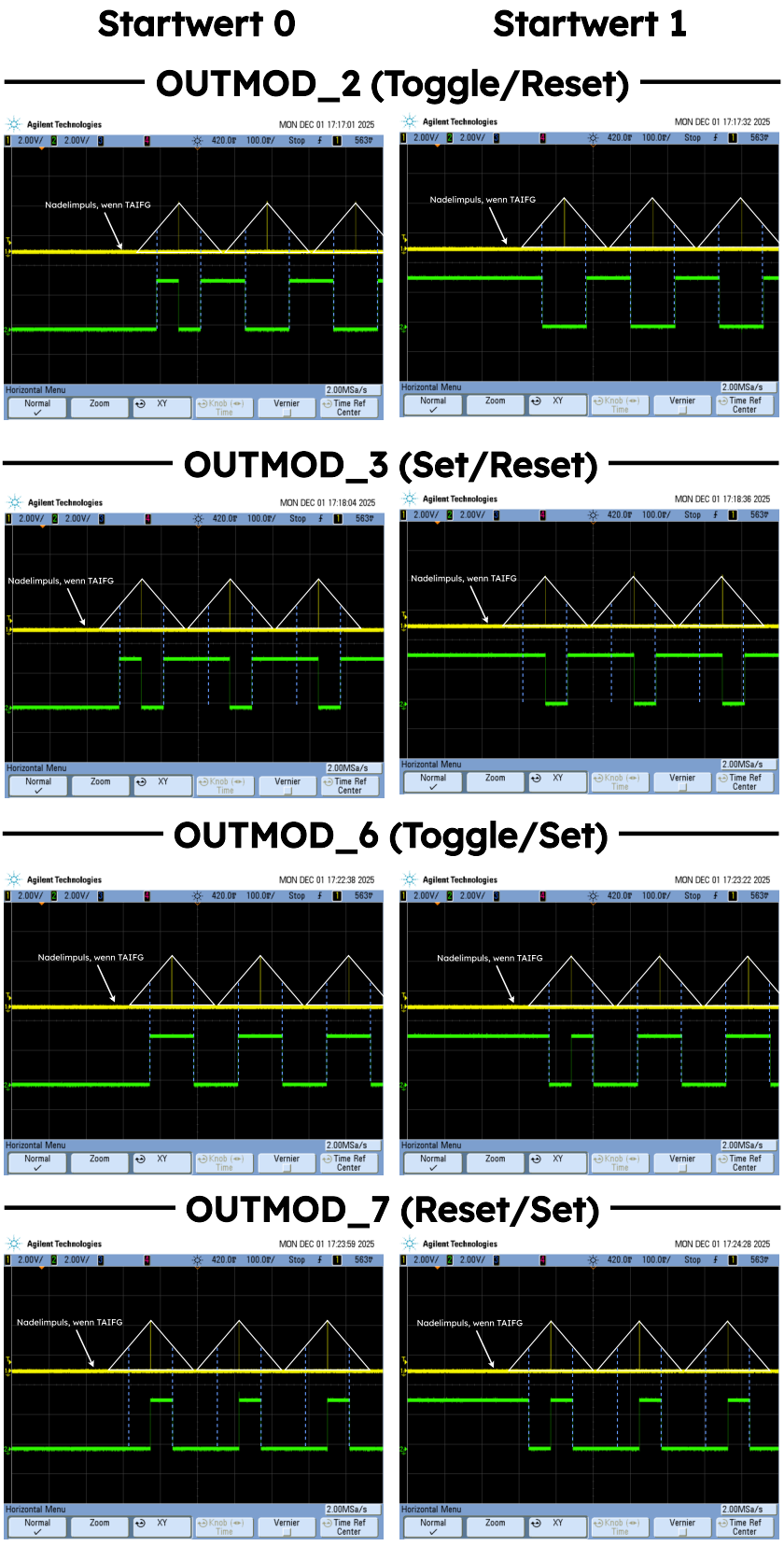
[12] Texas Instruments Incorporated. *MSP430F2xx, MSP430G2xx Family User's Guide*. Aug. 2022. URL: https://www.ti.com/lit/pdf/slau144k (besucht am 01. 12. 2025).

8 Anhang

Verzeichnis der Anhänge

```c
#include <msp430.h>

// OUTMOD_0, OUTMOD_1, OUTMOD_2, ... OUTMOD_7
#define TEST_MODE OUTMOD_7

// 0 = Start mit LOW (Simuliert die durchgezogene Linie in vielen Grafiken)
// 1 = Start mit HIGH (Simuliert die gestrichelte Linie)
#define START_LEVEL 1

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;   // Watchdog aus

    // --- PIN SETUP ---
    // P1.0 als Marker (Rote LED)
    P1DIR |= BIT0;
    P1OUT &= ~BIT0;
    // P1.6 als PWM-Ausgang (TA0.1 / Grüne LED)
    P1DIR |= BIT6;
    P1SEL |= BIT6;

    // --- TIMER SETUP (noch angehalten) ---
    // SMCLK, /8 Divider, Timer noch im STOP Mode (MC_0), Clear
    TA0CTL = TASSEL_2 | ID_1 | MC_0 | TACLR;

    // Periode (Spitze des Dreiecks)
    TA0CCR0 = 60000;
    // Umschaltpunkt für unser Signal (TA0.1)
    TA0CCR1 = 30000;

    // Pin in den gewünschten Startzustand zwingen (OUTMOD_0)
    if (START_LEVEL == 1) {
        TA0CCTL1 = OUTMOD_0 | OUT;
    } else {
        TA0CCTL1 = OUTMOD_0;
    }

    __delay_cycles(1000000);

    // Modus umschalten
    TA0CCTL1 = TEST_MODE;

    // Interrupt für CCR0 (TAIFG)
    TA0CCTL0 = CCIE;

    // Timer in Up/Down-Mode (MC_3) schalten und Interrupts (GIE) an
    TA0CTL |= MC_3;
    __bis_SR_register(LPM0_bits + GIE);
}

// Wenn Timer CCR0 (60000) erreicht -> Spitze
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A0_ISR(void) {
    // Kurzer Puls auf P1.0 als Trigger-Hilfe für das Oszi
    P1OUT |= BIT0;
    __delay_cycles(500); // Kleiner Delay für Sichtbarkeit
    P1OUT &= ~BIT0;
}
```

Anhang 1: *C-Code zum Messen des korrekten Ausgangsverhaltens: Der Ausgang von `TA0CCTL1` wird zunächst mit einem fixen Wert (`OUTMOD_0`) beschaltet, um nach einer Wartezeit ohne Timer-Reset einen uneingeschwungenen Zustand zu simulieren. `TAIFG` erzeugt eine Interrupt-Service-Routine, welche Pin `1.0` für kurze Zeit einschaltet, damit das Oszilloskop das Signal erkennt. Code KI-gestützt entwickelt.*



Anhang 2: *Messungen der Output-Modi 2, 3, 6 und 7 mit dem Oszilloskop. Die restlichen Modi waren in der Dokumentation von TI ohne Widersprüche. Es wird jeweils auch der uneingeschwungene Zustand simuliert, sodass auch die gestrichelten Linien simuliert werden können.*

```c
#include <msp430.h>

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;   // Watchdog aus

    // OUT0 (TA0.0) liegt auf P1.1!
    P1DIR |= BIT1;
    P1SEL |= BIT1; // P1.1 auf Timer-Funktion schalten

    // Oszilloskop triggern, wenn der Peak erreicht ist
    P1DIR |= BIT0;
    P1OUT &= ~BIT0;

    // SMCLK, /8 Divider, Up/Down Mode, Clear
    TA0CTL = TASSEL_2 | ID_3 | MC_3 | TACLR;

    // Periode (Spitze des Dreiecks)
    TA0CCR0 = 60000;

    TA0CCR1 = 30000;


    // CCIE, damit die ISR läuft
    TA0CCTL0 = OUTMOD_7 | CCIE;

    __bis_SR_register(LPM0_bits + GIE);
}

// läuft genau am Peak (bei 60000)
#pragma vector=TIMER0_A0_VECTOR
__interrupt void Timer_A0_ISR(void) {
    // Kurzer Puls auf P1.0 als Trigger-Hilfe für das Oszi
    P1OUT |= BIT0;
    __delay_cycles(5000); // Kleiner Delay für Sichtbarkeit
    P1OUT &= ~BIT0;
}
```

Anhang 3: *C-Code für das Testen der 'verbotenen' Modi für `CCR0` : Das Verhalten des Ausgangs von `TA0CCTL0` wird hier untersucht. Code KI-gestützt entwickelt.*