

Projekt USB-Oszilloskop

Samuel Oeser, Nicole Sturm, Daniel Wirth

22. September 2025

Inhaltsverzeichnis

1 Abstract / Zusammenfassung

2 Einleitung

Die vorliegende Projektarbeit wurde im Rahmen des Bachelorstudiengangs Elektrotechnik und Informationstechnik an der Fakultät für Elektrotechnik, Feinwerktechnik und Informationstechnik (EFI) der Technischen Hochschule Nürnberg Georg Simon Ohm durchgeführt. Ziel des Projekt-Moduls ist es, den Studierenden die Möglichkeit zu geben, ihr theoretisch erworbenes Wissen in einem praxisnahen, ingenieurwissenschaftlich strukturierten Entwicklungsprojekt anzuwenden. Die Motivation für die Auswahl des Projektthemas lag in der Abbildung des vollständigen Entwicklungsprozesses eines aus Hardware, Firmware und Software bestehenden Gesamtsystems. Auf diese Weise konnten praxisnahe Erfahrungen in allen wesentlichen Entwicklungsdisziplinen gesammelt werden. Darüber hinaus bot das Projekt die Gelegenheit, die grundlegenden Funktionen eines Oszilloskops zu verstehen und im Rahmen eines Prototyps zu realisieren. Ein weiteres wesentliches Auswahlkriterium für das Thema war die klare Unterteilung in abgegrenzte Aufgabenbereiche, sodass die Teammitglieder ihre Aufgaben eigenständig bearbeiten konnten, während gleichzeitig eine Zusammenarbeit im übergeordneten Kontext möglich war.

Das zu Beginn definierte Ziel des Projekts war die Entwicklung eines USB-Oszilloskops. Das Gesamtsystem, bestehend aus selbstentwickelter Hardware und einem über Universal Serial Bus (USB) angeschlossenen Computer, soll die Grundfunktionen eines digitalen Speicheroszilloskops (DSO) abbilden. Die Realisierung sollte durch den Einsatz eines Analog-Digital-Umsetzers (ADC) zur Messwerterfassung sowie eines Mikrocontrollers (μ C) als Schnittstelle zwischen der Hardware (ADC-Schaltung) und der Software (Computer) erfolgen.

Das Projektteam setzte sich aus drei Studierenden zusammen: Samuel Oeser war verantwortlich für die Entwicklung der Hardware (HW), Daniel Wirth übernahm die Firmware (FW), während Nicole Sturm die Software (SW) einschließlich der grafischen Benutzeroberfläche (GUI) entwickelte. Die Betreuung des Projekts erfolgte durch Prof. Dr. Sven Loquai, der die Studierenden während des gesamten Entwicklungsprozesses begleitete.

3 Fachliche Grundlagen

3.1 Allgemeiner Aufbau eines DSOs

Das DSO erfasst Eingangssignale, digitalisiert sie und stellt die Messdaten nach Speicherung und Verarbeitung dar. Der grundlegende Aufbau umfasst die Eingangsumschaltung (DC AC GND) mit Vertikalverstärkung/abschwächung, den ADU mit Signalvorverarbeitung, den Datenspeicher, Takt und Steuerung, die Triggereinrichtung sowie die Anzeige. Die digitalisierten Daten werden im Speicher abgelegt und für Darstellung und Auswertung ausgelesen (vgl. [Muehl2020]).

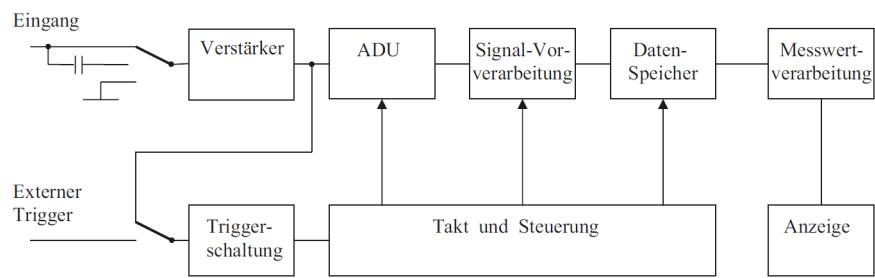


Abbildung 1: Blockschaltbild eines Digitaloszilloskops (Abb. 14.1 aus [Muehl2020])

Für die Abtastung sind zwei Betriebsarten relevant. Die Echtzeitabtastung erfassst den kompletten Verlauf in einem Durchlauf und die zeitliche Auflösung wird durch die Rate des AD-Umsetzers (ADU) begrenzt. Die Äquivalenzzeitabtastung rekonstruiert sehr schnelle periodische Verläufe aus mehreren Durchläufen (vgl. [Muehl2020]).

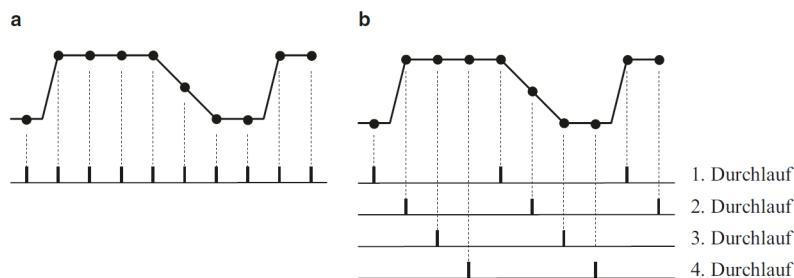


Abbildung 2: Gegenüberstellung Echtzeitabtastung (a) und Äquivalenzzeitabtastung (b) (Abb. 14.2 aus [Muehl2020])

Für eine gut auswertbare Darstellung sind etwa zehn Stützstellen je Periode zweckmäßig

und die si-Interpolation (auch sinc-Interpolation genannt) ermöglicht bei erfüllter Nyquist Bedingung eine nahezu verzerrungsfreie Rekonstruktion. In der Praxis sollte die maximale Signalfrequenz mindestens um den Faktor 2,5 unter der Abtastrate liegen (vgl. [Muehl2020]).

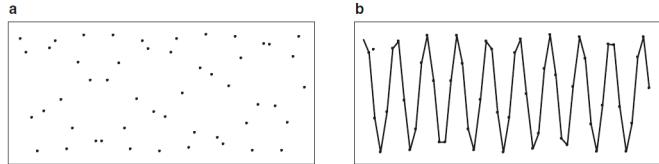


Abbildung 3: Punktendarstellung (a) und lineare Interpolation (b)
(Abb. 14.4 aus [Muehl2020])

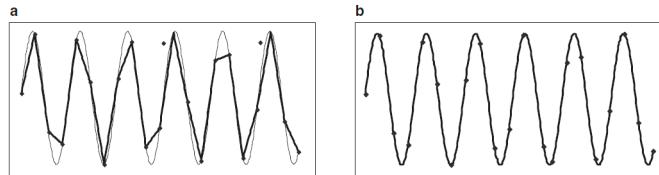


Abbildung 4: Lineare Interpolation (a) und si-Interpolation (b)
(Abb. 14.5 aus [Muehl2020])

In der Praxis werden für die Digitalisierung schnelle Flash-ADUs sowie mehrstufige Subranging- / Pipeline-Varianten eingesetzt. Diese ermöglichen hohe Raten bei moderater vertikaler Auflösung. Übliche Geräte arbeiten mit acht Bit vertikaler Auflösung. Je nach Architektur sind höhere Auflösungen möglich (vgl. [Bernstein2023]). Für die Datenaufnahme stehen verschiedene Acquisition-Modi zur Verfügung. Der direkte Modus liefert äquidistante Samples. Min-Max oder Peak-Detect erfasst Extremwerte innerhalb eines Abtastintervalls und macht kurze Spikes sichtbar. Average reduziert Rauschen durch wiederholtes Mitteln. Single Shot zeichnet einmalige Ereignisse auf und der Roll-Modus zeigt sehr langsame Verläufe kontinuierlich an (vgl. [Muehl2020]). Die Genauigkeit wird wesentlich durch die Abtastrate und Abtaststrategie, Interpolation und Quantisierung sowie durch die Eingangsbeschaltung bestimmt. Bei Unterabtastung droht Aliasing und die Rekonstruktion wird unzuverlässig (vgl. [Muehl2020], Grundlagen zur Abtastung und Quantisierung). Wichtige Kenngrößen von DSOs sind analoge Bandbreite, Abtastrate, vertikale Auflösung, Speichertiefe, Triggerfunktionen, Eingangsimpedanz, analoge

Eingangsspannung sowie Anzeige und Darstellungsoptionen. Über die Kurvendarstellung hinaus gehören auch automatische Messfunktionen zum Grundumfang.

3.1.1 Trigger

Die Triggerung steuert die Signalspeicherung. Fortlaufend erfasste Abtastwerte laufen in einen Ringspeicher und beim Triggerereignis wird das Überschreiben gestoppt, sodass Pre Trigger und Post Trigger gezielt darstellbar sind (vgl. [Muehl2020]).

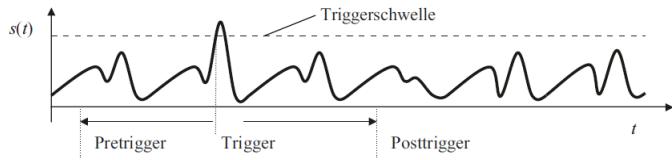


Abbildung 5: Messsignal $s(t)$ mit eingestellter Triggerschwelle und Triggerzeitpunkt
(Teilabbildung der Abb. 14.3 aus [Muehl2020])

Die analoge Triggerkette umfasst Quelle, Kopplung DC oder AC, einstellbare Triggerschwelle, Flankenrichtung, Hysterese und Auto Trigger. Sie erzeugt ein robustes Trigger-Signal für die Speichersteuerung (vgl. [Muehl2020]).

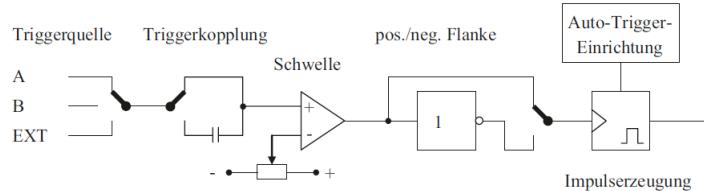


Abbildung 6: Funktionsblöcke einer Triggereinrichtung (Abb. 13.8 aus [Muehl2020])

3.1.2 Frequenzkompensierter Tastkopf / Spannungsteiler

In der Praxis existieren verschiedene Arten von Tastköpfen wie passive, aktive und differenzielle Tastköpfe. Im Folgenden wird ausschließlich der frequenzkompensierte passive Tastkopf betrachtet, da er für das Projekt relevant ist (vgl. [Muehl2020]). Dieser Tastkopf basiert auf einem frequenzkompensierten Spannungsteiler (siehe ??), realisiert als RC Teiler mit zu den Widerständen parallel geschalteten Kapazitäten und mit einer Anpassung an die Summe aus Bauteil- und Leitungskapazitäten. Die Kompensation mit

Trimmkondensator (in ??: C_T) wird so abgeglichen, dass das Teilungsverhältnis im Nutzfrequenzband annähernd konstant bleibt. Der Abgleich erfolgt mit einem Rechtecksignal. Unterkompensation zeigt sich durch eine nach unten geneigte Oberkante (??a), Überkompensation durch eine nach oben geneigte Oberkante (??c) und bei korrekter Kompensation bleibt die Oberkante über die Pulsbreite hinweg horizontal (??b) (vgl. [Schruefer2022]).

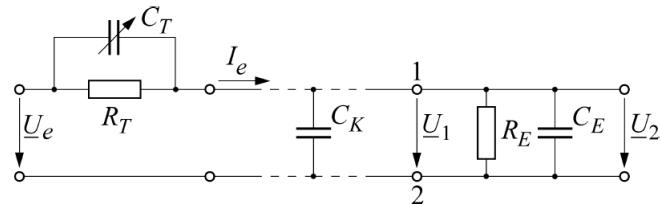


Abbildung 7: Tastteiler am Eingang eines Oszilloskops (Bild 2.42 aus [Schruefer2022])

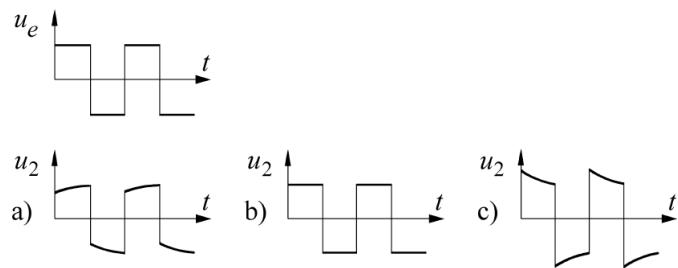


Abbildung 8: Rechteckimpulse an RC-Spannungsteiler (Bild 2.43 aus [Schruefer2022])
(a) unterkompensiert, (b) richtig kompensiert, (c) überkompensiert

Mit $R_1 = R_T$ und $C_1 = C_T$ gilt für die obere Teilimpedanz (\underline{Z}_1):

$$\underline{Z}_1 = R_1 \parallel \frac{1}{j\omega C_1} = \frac{R_1}{1 + j\omega R_1 C_1} \quad (1)$$

und mit $R_2 = R_E$ und $C_2 = C_K + C_E$ gilt für die untere Teilimpedanz (\underline{Z}_2):

$$\underline{Z}_2 = R_2 \parallel \frac{1}{j\omega C_2} = \frac{R_2}{1 + j\omega R_2 C_2} \quad (2)$$

Über den Spannungsteiler ergibt sich aus (??) und (??) das Verhältnis:

$$\frac{\underline{U}_1}{\underline{U}_2} = \frac{\underline{Z}_1 + \underline{Z}_2}{\underline{Z}_2} = 1 + \frac{\underline{Z}_1}{\underline{Z}_2} = 1 + \frac{R_1}{R_2} \frac{1 + j\omega R_2 C_2}{1 + j\omega R_1 C_1} \quad (3)$$

Dieses Verhältnis nimmt den frequenzunabhängigen Wert $\frac{U_1}{U_2} = 1 + \frac{R_1}{R_2}$ an, wenn gilt:

$$R_1 \cdot C_1 = R_2 \cdot C_2 \quad (4)$$

Durch Einsetzen der Einzelbauelemente für R_1 , R_2 , C_1 und C_2 und Auflösen nach C_T ergibt sich für den Trimmkondensator ein Wert nach ??, damit der Spannungsteiler frequenzunabhängig wird (vgl. [Schruefer2022]).

$$C_T = \frac{R_E \cdot (C_K + C_E)}{R_T} \quad (5)$$

3.2 AAF-Entwurf (Nyquisttheorem)

Ein Digitaloszilloskop erfasst das analoge Eingangssignal durch Abtastung mit einer bestimmten Frequenz. Diese Abtastung entspricht der Multiplikation des Signals mit einer Reihe von Impulsen, was im Frequenzbereich einer Faltung entspricht. Eine eindeutige Rekonstruktion aus den Abtastwerten setzt ein bandbegrenztes Eingangssignal mit höchster relevanter Frequenz und eine Abtastrate oberhalb des Doppelten dieser Frequenz voraus. Bei Unterschreitung falten sich Spektralkopien in das Basisband zurück und erzeugen Aliasse. Dies wird im Frequenzbild durch die Überlappung periodischer Spektren sichtbar in ?? sowie im Zeitbereich durch scheinbar niedrigere Schwingungen in ?. Aus dieser Grundlage folgt die Notwendigkeit eines vorgeschalteten Anti-Aliasing-Tiefpasses, der außerbandige Anteile vor der Wandlung ausreichend dämpft (vgl. [Bernstein2023] und [Bernstein2024]).

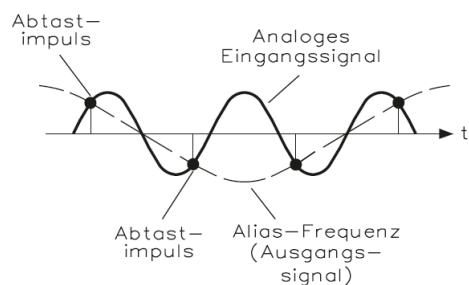


Abbildung 9: Aliasing: Erzeugung einer Scheinfrequenz durch unpassende Abtastrate
(Bild 2.71 aus [Bernstein2024])

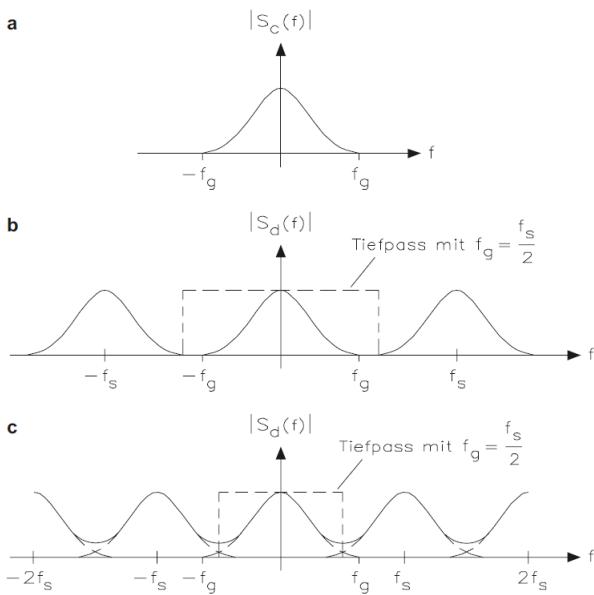


Abbildung 10: Abtasttheorem (Bild 4.30 aus [Bernstein2023])

- (a) Betragsspektrum des bandbegrenzten Signals
- (b) Vervielfachung des Basisspektrums durch Abtastung mit $f_s > 2 \cdot f_g$
- (c) Aliasing infolge zu niedriger Abtastrate ($f_s < 2 \cdot f_g$)

Da der Entwurf des Anti-Aliasing-Filters maßgeblich durch den ausgewählten AD-Umsetzer bestimmt wird, sollte dieser zuerst festgelegt werden. Für aktive RC-Filter bewährt sich wegen Bauteiltoleranzen und des nicht idealen Übergangs vom Durchlass- in den Sperrbereich eine Abtastrate oberhalb der Mindestbedingung, typischerweise etwa 2.5 bis 4 mal f_{max} und bei strengen Dämpfungszielen, z.B. wegen höheren Auflösungen, auch 5 bis 10 mal f_{max} . Eine höhere Abtastrate erweitert zudem die Übergangsbreite und senkt damit die notwendige Filterordnung, erleichtert die direkte Darstellung des abgetasteten Signals ohne Rekonstruktionsalgorithmen und erlaubt eine Filterfamilie mit geringerer Gruppenlaufzeitvariation und damit besserem Zeitverhalten. Aus der gewählten Abtastrate folgt dann die Nyquist-Frequenz $f_N = \frac{f_s}{2}$ und die Übergangsbreite $\Delta f = f_N - f_{max}$ [Pini2020].

Der theoretische, aus der Auflösung abgeleitete Dynamikbereich setzt eine Obergrenze für die sinnvoll anzustrebende Sperrbanddämpfung. Für einen N -Bit-Umsetzer mit Full-Scale-Range FSR gilt die Codebreite $Q = \frac{FSR}{2^N}$. Ein voll ausgesteuertes Störsignal soll auf einen Pegel unterhalb der Codebreite gedämpft werden, damit keine signifikanten Quantisierungsartefakte entstehen. Die Sperrbanddämpfung, beginnend an der Nyquist-Frequenz,

wird daher so festgelegt, dass aliasingverursachende Anteile unterhalb der Codebreite bleiben:

$$A(f_N)_{max} = 20 \cdot \log_{10} \left(\frac{FSR}{Q} \right) = 20 \cdot \log_{10}(2^N) = 6.02 \cdot N \text{ dB} \quad (6)$$

In der Praxis begrenzt das *SINAD* häufig den nutzbaren Dynamikbereich des gewählten AD-Umsetzers. Gilt $|SINAD| < |A_{max}|$ oder liegt die daraus ermittelte *ENOB* (siehe ?? “??“) unter der Auflösung des ADCs, dominieren Rauschen- und Verzerrungen. Bei Einsatz einer Mittelwertbildung kann es dennoch zweckmäßig sein, die Obergrenze A_{max} anzusetzen. Im Regelfall wird die Sperrbanddämpfung an der Nyquistfrequenz so gewählt, dass gilt (vgl. [Pini2020]):

$$A(f_N) \approx \min(|A_{max}|, |SINAD_{ADC}|) \quad (7)$$

Die Bewertung des entworfenen Filters erfolgt in einem Tool wie dem Analog Filter Wizard über die Diagramme von Amplitudengang, Phasen- und Gruppenlaufzeit sowie Sprung- und Impulsantwort. Der Frequenzgang belegt Passband-Flachheit, Übergangsbreite und die Dämpfung an f_N . Phase und Gruppenlaufzeit zeigen, ob das Filter im Nutzband annähernd eine konstante Verzögerung liefert und damit Impulse und Flanken formtreu bleiben. Die Sprungantwort offenbart Überschwinger und Einschwingzeit. Eine Toleranzanalyse der zu erwartenden Widerstands- und Kapazitätsstreuungen zeigt, ob die Reserven ausreichend dimensioniert wurden (vgl. [Pini2020] und [Padmanabhuni2017]).

3.3 Analog-to-Digital-Converter (ADC)

Ein Digitaloszilloskop benötigt einen AD-Umsetzer als zentrales Hardwareelement, da erst die Wandlung des analogen Eingangssignals in digitale Messwerte die weitere Speicherung, Verarbeitung und Anzeige ermöglicht. Für die Auswahl steht die Topologie am Anfang, weil sie Abtastrate, erreichbare Auflösung, Latenz, Leistungsaufnahme und den Implementierungsaufwand bestimmt.

Für ein USB-Oszilloskop bieten sich Flash- und Pipeline-Topologien an, da beide hohe Abtastraten für breitbandige Zeitbereichsmessungen bereitstellen. Andere Varianten wie SAR oder Delta-Sigma werden in den Quellen überwiegend mittleren Geschwindigkei-

ten oder schmalbandigen Präzisionsaufgaben zugeordnet und sind somit für ein USB-Oszilloskop irrelevant (vgl. [MPS] und [AnalogDevices2001a]).

3.3.1 Flash-ADC

?? zeigt das Prinzip eines Flash-AD-Umsetzers mit zwei Bit. Eine Widerstandsleiter erzeugt abgestufte Referenzen und eine Komparatorbank vergleicht das Eingangssignal mit diesen Schwellen. Das resultierende Muster wird durch eine Logikschaltung in einen Binärcode überführt. Flash-ADCs erreichen, durch diesen simplen Aufbau, die höchste Geschwindigkeit. Da die Zahl der Komparatoren jedoch $2^N - 1$ beträgt, steigen bei hohen Auflösungen Komplexität, Fläche, Leistungsaufnahme sowie insbesondere die Herstellungskosten annähernd exponentiell an (vgl. [MPS] und [AnalogDevices2001b]).

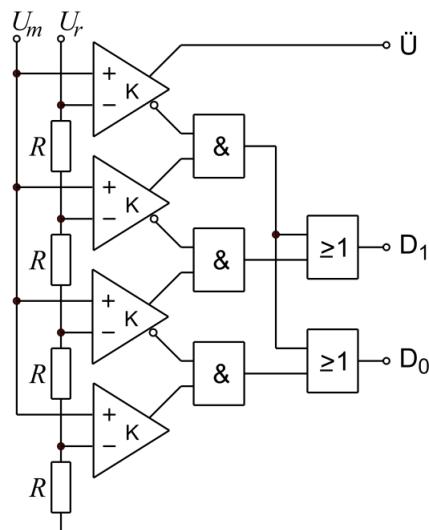


Abbildung 11: Flash-Umsetzer mit 2 Bit, einschl. Kodeumsetzer
Ersteller: Saure, Quelle: [\[Link\]](#) (Lizenz: CC BY-SA 3.0)

3.3.2 Pipeline-ADC

Eine Alternative zur geringeren Skalierbarkeit des Flash-ADCs ist der Pipeline-ADC, der auch im Projekt zur Anwendung kommt. Er baut funktional auf dem Flash-Prinzip auf, indem mehrere niedrigauflösende Flash-ADCs in Stufen hintereinandergeschaltet werden. Am Eingang tastet eine Sample-and-Hold-Stufe das Signal ab. Jede Stufe wandelt in we-

nige Bits, rekonstruiert den quantisierten Anteil über einen DAC, bildet den Restfehler im Subtrahierer und verstärkt diesen für die nächste Stufe. Durch Überlappbits und digitale Fehlerkorrektur können die Genauigkeitsanforderungen an die einzelnen Stufen reduziert werden und die Durchsatzraten erhöht. Durch diesen Aufbau steigt der Aufwand mit zunehmender Auflösung nur linear an. Bauartbedingt entsteht allerdings eine definierte Zykluslatenz, die in ganzen oder halben Taktzyklen angegeben wird. Diese Latenz ist der entscheidende Nachteil eines Pipeline ADCs (vgl. [MPS] und [AnalogDevices2001c]).

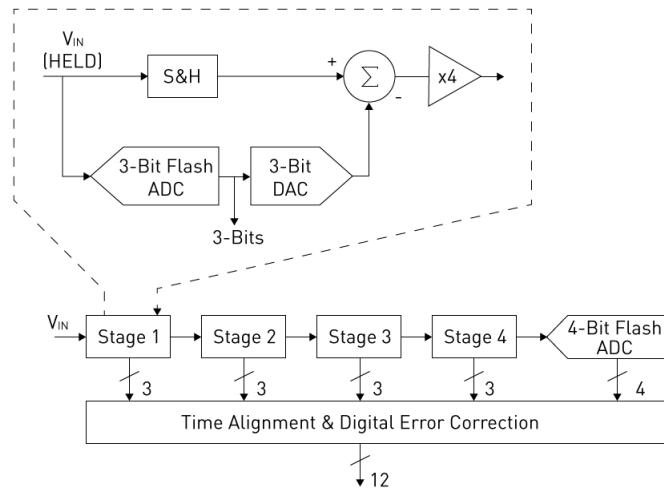


Abbildung 12: 12-bit pipelined ADC
Ersteller: MPS, Quelle: [\[Link\]](#) (Ch. 2, Fig. 4)

3.3.3 Wichtige Kenngrößen von AD-Umsetzern

Die Kenngrößen und deren Definition können dem IEEE Standard 1241-2023 “*Terminology and Test Methods for Analog-to-Digital Converters*“ ([IEEE2023]) entnommen werden.

Auflösung und Eingangsbereich

Die Auflösung eines N -Bit-AD-Umsetzers umfasst 2^N binäre Codes. Der differentielle Eingangsspannungsbereich wird als Full-Scale Range (FSR) bezeichnet und in Volt angegeben. Das LSB entspricht $\frac{FSR}{2^N}$.

Maximale Sampling-Rate

Die Abtastrate ist der Kehrwert der Abtastperiode (Abstand zwischen Abtastwerten). Sie bestimmt die zeitliche Auflösung und ist insbesondere in Bezug auf das Niyquisttheorem relevant.

Leistungsaufnahme

Die Leistungsaufnahme ist in erster Linie für die Auslegung der Versorgungsspannung und der zugehörigen Leitungen relevant.

Verstärkungsfehler

Der Verstärkungsfehler beschreibt die Abweichung der Kennliniensteigung von der idealen Steigung nach Offsetkorrektur und wird in LSB oder in Prozent der Full-Scale-Range angegeben.

Offsetfehler

Der Offsetfehler ist die vertikale Verschiebung der Kennlinie relativ zur Referenzgeraden und wird in LSB oder in Prozent der Full-Scale-Range angegeben.

Latenz

Die Latenz wird als Pipeline-Delay in ganzen oder halben Taktzyklen angegeben oder alternativ als Zeitangabe und beschreibt den Abstand zwischen Abtastung und Verfügbarkeit des zugehörigen Digitalworts.

Differentielle Nichtlinearität DNL

Die DNL ist die Abweichung der tatsächlichen Codebreite $W[k]$ von der idealen Codebreite Q , normiert auf Q , und wird in LSB angegeben. Die Codebreite entspricht dem Spannungsbereich, der dem Code zugeordnet ist. Formal gilt $DNL[k] = \frac{W[k]}{Q} - 1$. Die Angabe im Datenblatt entspricht $\max DNL[k]$.

Integrale Nichtlinearität INL

Abweichung des Übergangsspannungsniveaus von einem Code in den nächsten zu dem der idealisierten Geraden, nach Korrektur von Offset und Verstärkung.

Damit gilt $INL[k] = \sum_{n=0}^k DNL[n]$. Die Angabe im Datenblatt entspricht $\max INL[k]$.

Rausch- und Verzerrungsmaße

SNR gibt das Verhältnis von Nutzsignal- zu Rauschleistung an:

$$SNR_{dB} = 10 \cdot \log_{10} \left(\frac{P_{Signal}}{P_{Rauschen}} \right) \quad (8)$$

THD ist ein Maß für das Verhältnis von harmonischen Verzerrungen zur Grundschwingung:

$$THD_{dB} = 10 \cdot \log_{10} \left(\frac{P_{Harmonische}}{P_{Signal}} \right) \quad (9)$$

$SINAD$ umfasst harmonische Verzerrungen und Rauschen:

$$SINAD_{dB} = 10 \cdot \log_{10} \left(\frac{P_{Signal}}{P_{Rauschen} + P_{Harmonische}} \right) \quad (10)$$

$ENOB$ ist die effektive Bitzahl und wird aus SINAD abgeleitet:

$$ENOB \approx \frac{SINAD[dB] - 1.76}{6.02} \quad (11)$$

3.4 Endliche Zustandsautomaten (Finite State-Machines - FSMs)

Die Firmware und Software des Projekts sind als synchronisierte Zustandsautomaten implementiert, um einen deterministischen Ablauf der beiden Programme zu gewährleisten.

3.4.1 Definition und formale Darstellung

Ein endlicher Automat (EA - engl. Finite State Machine, FSM) ist ein abstraktes Rechenmodell zur Beschreibung von Systemen. Dieser befindet sich in mindestens einem Zustand von einer Zahl endlicher Zustände. Zustandsübergänge (sog. Transitionen) erfolgen durch Eingaben oder das Auftreten von Ereignissen (spezielle Form der Eingabe).

Zustände modellieren, was das System gerade tut bzw. in welchem internen “Modus“ es sich befindet. Ereignisse sorgen für den Wechsel zwischen Zuständen. Übergänge definieren, wie das System im aktuellen Zustand auf ein Ereignis reagiert.

Ein FSM besteht nach [Baesig2019] typischerweise aus:

- einer Menge von Zuständen S ,
- einem Anfangszustand s_0 ,
- einem Eingabe- oder Ereignisalphabet E (Events),
- einem Ausgabealphabet A ,
- einer Zustandsübergangsfunktion $\delta : S \times E \rightarrow S$,
- einer Ausgabefunktion $\lambda : S \times E \rightarrow A$,
- und einer endlichen (evtl. leeren) Menge der Endzustände F .

Je nach Modellierung kann das Ausgabealphabet und die Ausgabefunktion, sowie die Menge an Endzuständen entfallen.

Unterschieden werden folgende Arten von Endlichen Automaten:

- *Deterministische Endliche Automaten (DEA):*

Ein Automat wird als deterministisch bezeichnet, wenn für jede Kombination aus Eingabedaten und aktuellem Zustand eindeutig festgelegt ist, welcher Folgezustand erreicht wird. Das bedeutet, dass sich das *System zu einem bestimmten Zeitpunkt stets in genau einem definierten Zustand* befindet.

- *Nichtdeterministische Endliche Automaten (NEA):*

Bei einem NEA sind für einen bestimmten Zustand und eine Eingabe keine, genau ein oder mehrere mögliche Zustandsübergänge definiert.

DEAs sind einfacher als NEAs zu implementieren, wodurch diese in der Praxis häufiger zur Anwendung kommen (vgl. [Baesig2019]). Im Projekt sind die FSMs als DEAs ausgeführt .

Eine weitere Aufgliederung von DEAs erfolgt nach der Abhängigkeit der Ausgabe, wenn diese vorhanden ist (vgl. [Baesig2019]).

- Wenn die Ausgabe nur vom aktuellen Zustand abhängt, handelt es sich um einen *Moore-Automaten*.
- Wenn die Ausgabe vom aktuellen Zustand und der Eingabe abhängt, handelt es sich um einen *Mealy-Automaten*.

Im Projekt kommen Moore-Automaten zur Anwendung.

3.4.2 Modellierung und Darstellung

Die Darstellung eines Zustandsautomaten erfolgt üblicherweise mit einem *Zustandsdiagramm* (auch Zustandsübergangsdiagramm), einem gerichteten Graph¹. Die Knoten stellen die Zustände der Menge S dar und die gerichteten Kanten die Zustandsübergänge. Der Startzustand und die Endzustände werden gesondert gekennzeichnet. Eine übliche Darstellung für einen einfachen Moore-Automaten findet sich in ??a (sn sind die Zustände, xn sind Eingaben oder Ereignisse und yn sind Ausgaben; $n = 1, 2, \dots$) (vgl. [Baesig2019]).

Die Darstellung des Zustandsdiagramms ist auch als Teil der *Unified Modelling Language (UML)*² festgelegt, welche auch durch internationale Normung festgeschrieben ist. In der Praxis (vor allem beim händischen Entwurf von Zustandsdiagrammen) hat sich aber ein Mischform bewährt, welche auf einige Elemente der UML zurückgreift (siehe ??b). Es lassen sich beispielsweise auch hierarchische Strukturen integrieren (verschachtelte Zustände).

¹“Graphen sind ein [...] Konzept, um Objekte und die Beziehung zwischen diesen darzustellen.”
Quelle: [\[Link\]](#)

²aktueller UML-Standard (V2.5.1): [\[Link\]](#)

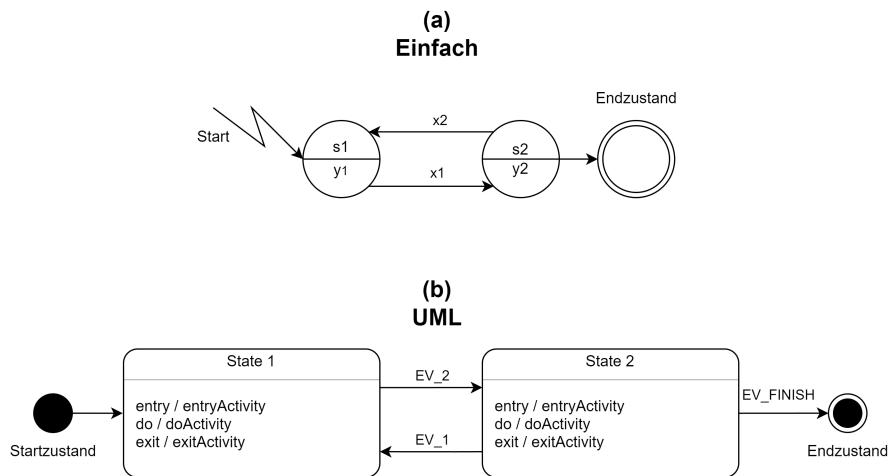


Abbildung 13: (a) Zustandsdiagramm (nach [Baesig2019]); (b) UML-Zustandsdiagramm

3.4.3 Vorteile einer FSM bei der Programmierung

- *Klarheit & Übersichtlichkeit*

Der Systemverlauf ist in wohldefinierte Zustände gegliedert. Der Code wird hierdurch verständlicher (kein “Spaghetti-Code“).

- *Deterministisches Verhalten*

Durch die Definitionen, wie auf welches Ereignis reagiert wird, kann die Vorhersagbarkeit und Zuverlässigkeit gewährleistet werden. Außerdem lassen sich undefinierte Zustände durch die explizite Modellierung vermeiden.

- *Modularisierung & Wiederverwendbarkeit*

Einzelne Zustände oder Teile der FSM lassen sich kapseln, segmentieren und wiederverwenden. Teilbereiche können eventuell separat getestet werden.

- *Wartbarkeit & Erweiterbarkeit*

Das System lässt sich durch Hinzufügen neuer Zustände oder neuer Übergänge modular erweitern, ohne die vorhandene Logik stark zu verändern.

- *Kommunikation und Dokumentation*

Zustandsdiagramme können bei der Vermittlung von komplexem Programmverhalten gegenüber Nicht-Programmierern genutzt werden.

- *Verbessertes Debugging*

Da Zustandsübergänge zentralisiert sind, lassen sich Log-Messages und weitere Debugging-

Mechanismen leichter verwenden.

3.5 Direct Memory Access - DMA

3.5.1 Allgemeines

DMA bezeichnet den Vorgang, bei dem durch eine dedizierte Einheit ohne Beteiligung einer CPU Daten transferiert werden (also als Hintergrundprozess). Er kommt zum Einsatz, wenn große Datenmengen von der Peripherie oder einem Speicher zu einem anderen transferiert werden müssen. Ein Chip oder eine Teileinheit eines Mikrocontroller (μ C), die solche Transfers durchführt, heißt *Direct Memory Access Controller (DMAC)* (vgl. [Urbaneck2020]). Im Folgenden soll die Funktionsweise anhand der Realisierung in der im Projekt verwendeten STM32-Mikrocontrollerfamilie erläutert werden.

3.5.2 DMA bei STM32-Mikrocontrollern

Der DMA-Controller ist ein AHB-Modul (AHB - advanced high-performance bus; standardisiertes Bussystem von ARM) und kann wie auch die CPU des μ Cs als Master auf diesen Bus (genauer auf die Bus-Matrix) zugreifen. Durch die Datentransfers, welcher der DMAC nach seiner Konfiguration ohne CPU-Beteiligung ermöglicht, kann die System-Performance deutlich erhöht werden. Die Datentransfers können hierbei per Software oder über angeschlossene Peripherieelemente über sog. *Requests* gestartet werden. Wird als steuerndes Element beispielsweise ein Hardware-Timer genutzt ermöglicht dies die zeitlich exakte Taktung von Datentransfers [STmicroelectronics2016], was im Projekt für den Abtastvorgang genutzt wird.

Der μ C besitzt zwei unabhängige DMAC (Aufbau siehe ??), deren Anbindung an Peripherie und Speicher unterschiedlich ausfällt, um alle Ressourcen des Systems flexibel zu verwenden.

Der DMAC besitzt 3 Schnittstellen:

- *slave port* zur Programmierung des DMAs
- *2 master ports*
 - *peripheral port*: periphereseitiger Datenanschluss
 - *memory port*: speicherseitiger Datenanschluss

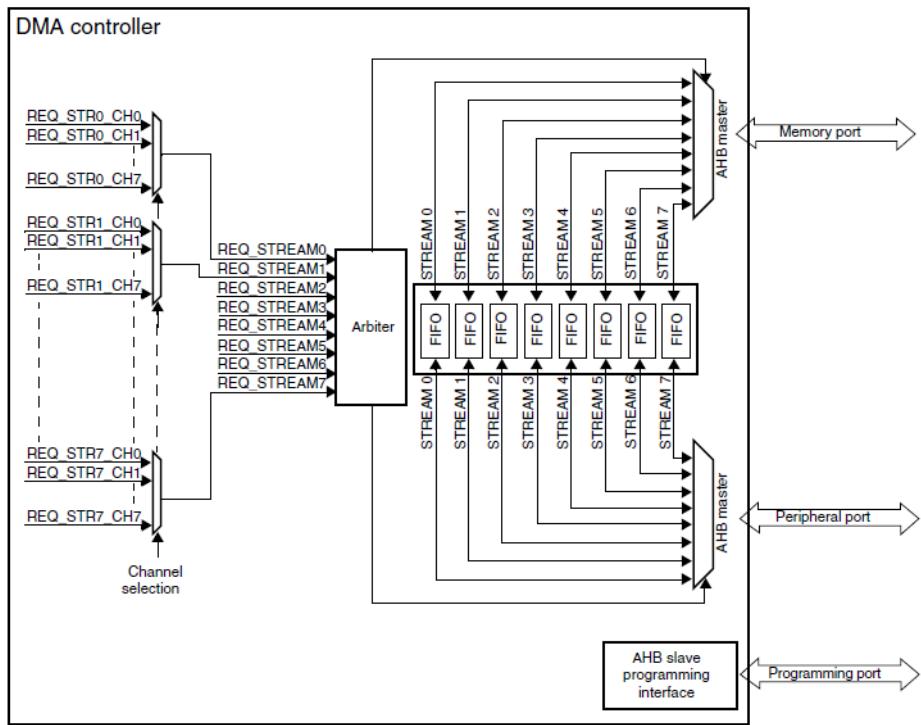


Abbildung 14: Blockdiagramm STM32-DMA-Controller aus [STmicroelectronics2016]

Jeder Controller besitzt 8 Datenwege (*Streams*), die separat für unterschiedliche Datentransfers konfiguriert werden können (die Transfers können aber nicht gleichzeitig laufen). Die Streams besitzen hierbei eine konfigurierbare Priorität und ein zentrales Modul, der *Arbiter*, regelt den Zugriff der Streams auf die Ports in Abhängigkeit der Priorität und sorgt für einen deterministischen Ablauf der Transfers. Die einzelnen Streams besitzen noch eine Anzahl an *Channels*, über die der entsprechende Peripherie-Request für einen Stream ausgewählt werden kann (vgl. [STmicroelectronics2016]). Die Zuordnung eines Peripherie-Requests zu einer Channel-Stream-Kombination kann dem Reference-Manual des Controllers entnommen werden (für den STM32F767: [STmicroelectronics2024]).

Jeder Stream besitzt außerdem einen 4-stufige *FIFO*-Pufferspeicher (First-In-First-Out), welcher Latenzen beim Zugriff auf das Übertragungsmedium (Bus-Switch-Matrix) überbrücken kann und ein *Verpacken/Entpacken der Daten* erlaubt (z.B.: input: 8bit-Pakete, output: 32bit-Pakete; siehe ??).

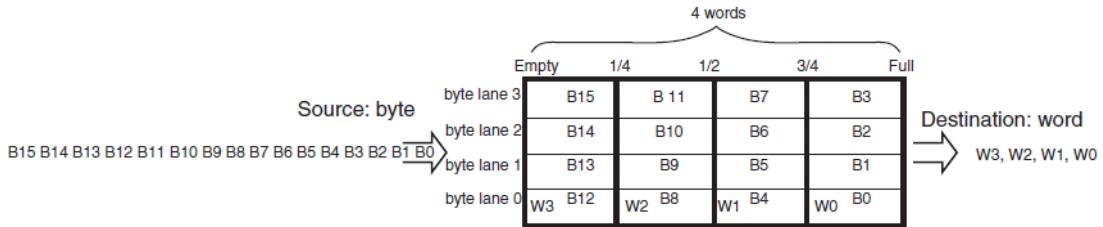


Abbildung 15: FIFO-Struktur aus [STmicroelectronics2016] (Teilabbildung)

DMA-Transfers

Ein Transfer wird zunächst über die *Quelladresse (source address)* und *Zieladresse (destination address)* charakterisiert, hier kann der DMA so konfiguriert werden, dass die Adressen nach einem Transfer automatisch inkrementiert werden können. Somit lassen sich, im Speicher hintereinanderliegende, Daten einfach übertragen. Quell- oder Zieladresse können jeweils aber auch konstant gehalten werden (siehe ??).

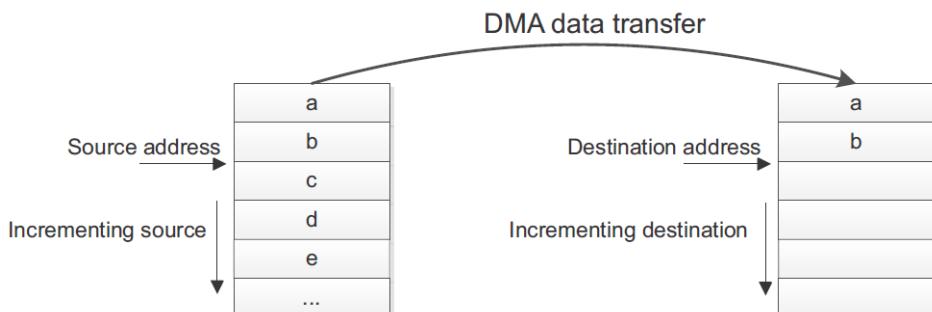


Abbildung 16: DMA Quell- und Zieladressinkrementierung aus [STmicroelectronics2016]

Weitere wichtige Parameter sind die Übertragungsgröße (transfer size), die in einem dedizierten Register (*NDTR* - Number of Data Transfers Register) abgelegt ist, und die Datenbreite (Byte, Half-word, Word). Der Inhalt des NDTR wird nach jedem Transfer entsprechend der Größe des Transfers dekrementiert. Hier wird noch zwischen Circular

mode und Normale mode unterschieden. Beim Normal mode ist eine Transaktion (bestehend aus NDTR Transfers) bei NDTR=0 beendet. Der Stream wird deaktiviert und es finden bis zum nächsten Aktivieren des Streams keine Transfers mehr statt. Beim Circular mode wird bei NDTR=0 das Register NDTR mit dem Initialwert geladen und die Transfers beginnen erneut (auch die Adressregister werden mit den Initialwerten geladen) → Kreislauf (circular).

Die drei möglichen Transfer-Modi sind:

- Peripheral-to-Memory (siehe Beispiel in ??))
- Memory-to-Peripheral
- Memory-to-Memory

Bei einem Timer-Überlauf des Hardware-Timers TIM1 findet ein DMA-Request statt (ein Stream wurde hierbei entsprechend konfiguriert). Dieser Request löst einen Daten-Transfer zwischen Ziel und Quelle aus (die Datenbreite wurde im Beispiel auf 1 Byte festgelegt). Anschließend werden die Adressen entsprechend der Datenbreite inkrementiert, um die Ziel- und Quelladresse für den nächsten Transfer vorzubereiten. Dieser Prozess geschieht so lange, bis “Number of Transfers“ durchgeführt worden sind. Der Wert von NDTR wird nach dem Transfer dekrementiert.

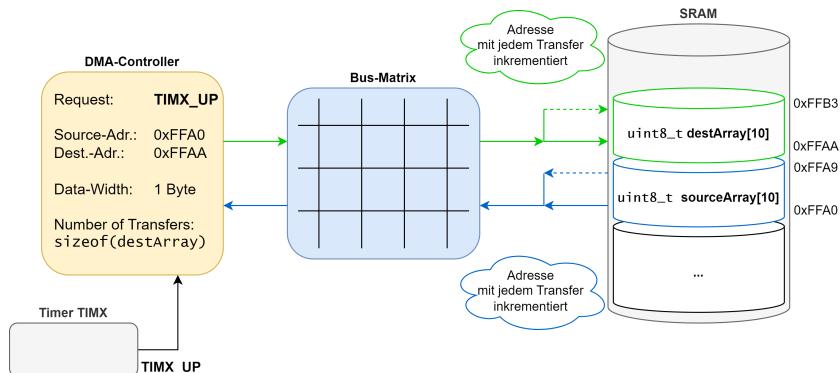


Abbildung 17: peripheral-to-memory-Transfer

Im Abtastsystem ist keine Inkrementierung der Quelladresse notwendig, da nur das Eingangsdatenregister ausgelesen werden muss, welches die Schnittstelle zum parallelen ADC-Interface darstellt (siehe ?? im Entwurfsteil der FW).

DMA-Transferwege

Gemeinsames Übertragungsmedium ist die Bus-Switch-Matrix, die auch in den vorherigen Abbildungen schon dargestellt wurde. Der Zugriff auf diese wird mit Hilfe einer Arbitrierung nach einem round-robin-Algorithmus geregelt. Durch den Arbitrierungsvorgang oder die Blockierung des Übertragungsmediums durch einen anderen Bus-Master (z.B. die CPU) kann eine *Latenz beim DMA-Transfer* entstehen.

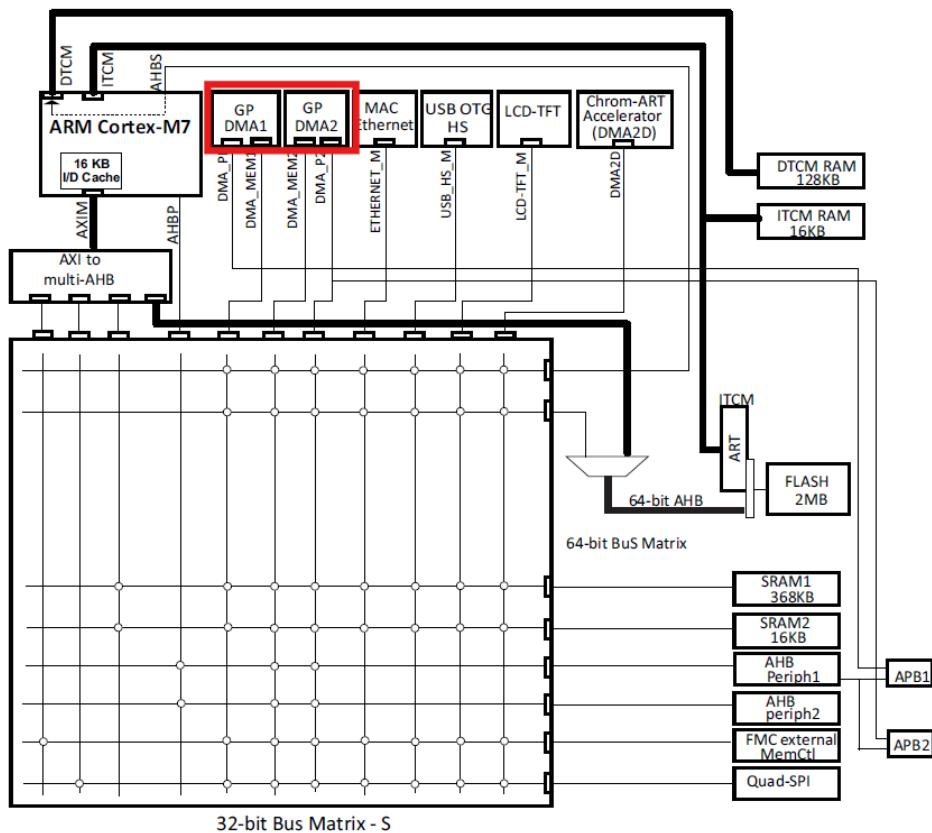


Abbildung 18: Systemarchitektur für STM32F76xxx μC aus [STmicroelectronics2024]

Nur Bus-Master (z.B.: Prozessor, DMA-Controller (in ?? rot hervorgehoben)) können Lese- und Schreiboperationen initiieren.

3.6 Digitale Filterung (Preprocessing)

3.6.1 Motivation und Grundlagen digitaler Filterung

Digitale Filterung ist ein zentrales Element der digitalen Signalverarbeitung (DSP). Filter werden im Allgemeinen für zwei Hauptzwecke eingesetzt:

- Trennung von Signalen, die miteinander überlagert wurden,
- Wiederherstellung von Signalen, die auf ihrem Übertragungsweg oder bei der Messung verzerrt wurden [Smith1999].

Während analoge Filter durch elektronische Schaltungen realisiert werden, bieten digitale Filter eine wesentlich höhere Präzision und Flexibilität. Sie können sehr steile Übergänge zwischen Durchlass- und Sperrbereich (passband und stopband) erzeugen und zeigen kaum Toleranzprobleme wie analoge Filter. Beim analogen Filterdesign ist die Beherrschung von Bauteiltoleranzen von wesentlicher Bedeutung [Smith1999]. Digitale Filterung bietet sich bei Anwendungen mit Mikrocontrollern an, da die Filter-Algorithmen direkt einprogrammiert werden können und eine analoge Filterschaltung ersetzen oder reduzieren können.

3.6.2 Filterklassifikation

Einteilung nach Signalcharakteristik

Die Einteilung der Filter richtet sich einerseits danach, in welcher Form die Information im Signal enthalten ist [Smith1999].

- *Zeitbereich (time-domain):*

Die Information ist direkt in der Form des Signals enthalten.

Bsp.: Glätten von Messdaten, DC-Offset-Entfernung, Signalerkennung.

→ Die Sprungantwort (step-response) ist hier entscheidend.

- *Frequenzbereich (frequency-domain):*

Die Information steckt in den Frequenzkomponenten des Signals.

Bsp.: Trennung von Audiosignalen oder Herausfiltern bestimmter Störfrequenzen.

→ Der Frequenzgang (frequency-response) ist hier entscheidend.

Einteilung nach Implementierung

Die Einteilung der Filter richtet sich einerseits danach, in welcher Form die Information im Signal enthalten ist [Smith1999].

- *FIR-Filter (Finite Impulse Response):*

Umsetzung erfolgt durch **Faltung (Convolution)** des Eingangssignals mit einem endlichen Filterkern (*engl. filter kernel*). Ein Ausgangswert wird berechnet, indem Eingangswerte mit zuvor definierten Gewichtungsfaktoren multipliziert und anschließend aufaddiert werden.

Eigenschaften:

- Stabil und vorhersagbar,
- Endliche Impulsantwort,
- Einfach zu implementieren, aber rechenintensiv.
- Beispiel: Moving Average Filter.

- *IIR-Filter (Infinite Impulse Response):*

Umsetzung durch **Rekursion**, bei der sowohl aktuelle Eingangswerte als auch frühere Ausgangswerte zur Berechnung herangezogen werden.

Eigenschaften:

- Sehr effizient (weniger Rechenaufwand),
- Theoretisch unendliche Impulsantwort,
- Gefahr der Instabilität bei fehlerhafter Parametrierung.
- Beispiel: digitale Tiefpassfilter auf Basis analoger Schaltungen.

3.6.3 Gleitender Mittelwertfilter

Der *gleitende Mittelwertfilter* (engl. *Moving Average Filter - MAF*) ist einer der einfachsten und am häufigsten eingesetzten digitalen Filter. Er arbeitet, indem er für jedes Ausgabesample den Mittelwert aus einer festen Anzahl aufeinanderfolgender Eingangssamples berechnet [Smith1999]. Seine Hauptaufgabe besteht darin, zufälliges Rauschen zu reduzieren (siehe ??), während plötzliche Signaländerungen (Steps) möglichst gut erhalten bleiben.

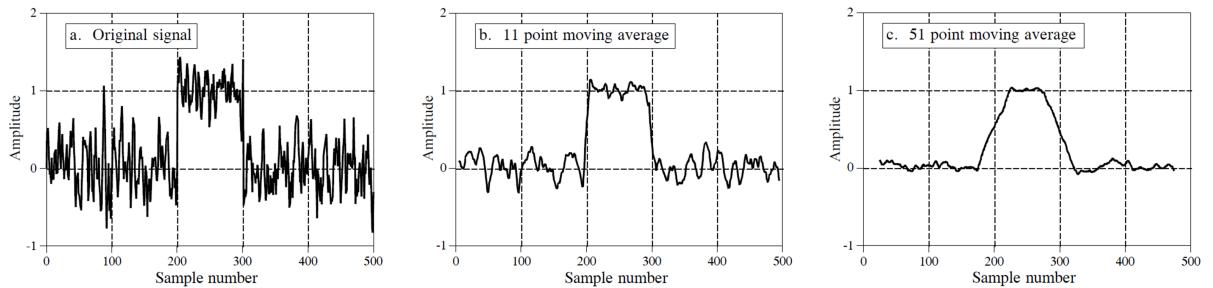


Abbildung 19: Anwendung eines MAF unterschiedlicher Fensterbreite
 (a) Eingangssignal, (b) Ausgangssignal $M = 11$, (c) Ausgangssignal $M = 51$
 (Figure 15-1 aus [Smith1999])

Grundidee Für ein Fenster der Breite M ergibt sich das Ausgangssignal $y[i]$ aus dem Eingangssignal $x[i]$ durch:

$$y[i] = \frac{1}{M} \sum_{j=0}^{M-1} x[i+j] \quad (12)$$

Es ist zu erkennen, dass die rechtsseitigen Eingangswerte zum Ausgangswert aufsummiert werden. Eine alternative Umsetzung wäre es die aufzusummierenden Eingangswerte symmetrisch um den Ausgangswert zu wählen (Beispiel für einen Ausgangswert):

$$y[80] = \frac{x[78] + x[79] + x[80] + x[81] + x[82]}{5} \quad (13)$$

Damit entspricht der Moving Average Filter einer Faltung des Eingangssignals mit einem rechteckigen Filterkern [Smith1999].

Für das Projekt wurde der Filter kumulativ implementiert und nach dem Algorithmus in ?? im ?? umgesetzt.

4 Projektkonzeption

4.1 Vorgehensweise

Das übergeordnete Vorgehen folgte einem strukturierten Prozess, der sich gut durch das **V-Modell** visualisieren lässt. Diese Vorgehensweise eignete sich insbesondere aus drei Gründen: *Erstens*, Änderungen an Hardware sind nach deren Umsetzung nur schwer möglich, was iterative Prozesse sehr aufwendig macht. *Zweitens* standen die Projektziele von Beginn an eindeutig fest. *Drittens* waren die Aufgabenbereiche klar voneinander abgegrenzt. Zudem handelte es sich um ein überschaubares Projekt.

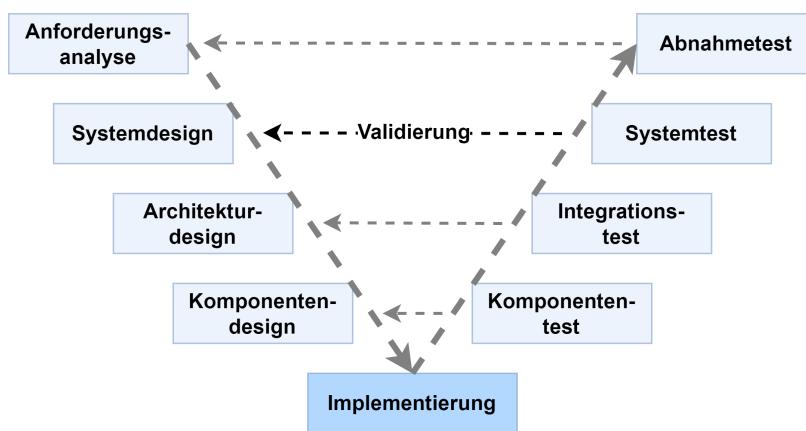


Abbildung 20: V-Modell

Im Folgenden werden die einzelnen Phasen und zugehörigen Validierungsschritte kurz erläutert.

Anforderungsanalyse und Abnahmetest

In enger Abstimmung mit dem betreuenden Professor wurden die grundlegenden Anforderungen definiert. Dazu gehörten unter anderem Bandbreite, Abtastrate, Auflösung und Eingangsspannungsbereich. Der spätere Abnahmetest bestand darin zu überprüfen, ob das entwickelte System diese Vorgaben in einer realistischen Versuchsumgebung erfüllte.

Systemdesign und Systemtest

Anschließend wurden die grundlegenden Systemfunktionen festgelegt, die zur Erfüllung der Anforderungen notwendig waren. Dazu zählten Signalaufnahme, Digitalisierung, Weiterleitung über USB und Visualisierung am PC. Im Systemtest wurde das fertige Gerät als Gesamtsystem geprüft.

Architekturdesign und Integrationstest

In dieser Phase erfolgte die Aufteilung des Systems in die Bereiche Hardware, Firmware und Software. Dabei wurden die Schnittstellen definiert, die ermöglichen sollten, die Module weitgehend unabhängig voneinander zu entwickeln. Diese wurden während des Komponentendesigns bei Bedarf konkretisiert. Im Integrationstest wurde die Zusammenarbeit dieser Module überprüft, insbesondere die zuverlässige Kommunikation zwischen den Subsystemen.

Komponentendesign und Komponententest

In dieser Phase wurden die Teilsysteme detailliert ausgearbeitet. Es wurden Konzepte erstellt, Bauteile gewählt, Implementierungsmöglichkeiten abgewogen etc. Jedes Subsystem wurde einzeln getestet, z. B. Spannungsversorgung und ADC in der Hardware, Speicherzugriffe in der Firmware oder Datenanzeige in der Software.

Implementierung

In der zentralen Phase wurden die geplanten Komponenten umgesetzt. Dies umfasste das Layout der Schaltungen, die Programmierung der Firmware sowie die Entwicklung der PC-Software. Ziel war es, die entworfenen Module funktionsfähig zu realisieren und für die Integration bereitzustellen.

Organisation und Zusammenarbeit

Die Zusammenarbeit erfolgte hauptsächlich über ein gemeinsames Projektteam in Microsoft Teams. Dieses war in mehrere Kanäle gegliedert: einen allgemeinen Kanal für projektübergreifende Informationen, jeweils eigene Kanäle für Hardware, Firmware und Software sowie zusätzliche Kanäle für die Schnittstellen. Diese Struktur erleichterte die Ablage von Quellen, Materialien und Informationen und stellte die Übersicht über das gesamte Projekt sicher.

Die Abstimmung innerhalb des Teams erfolgte in regelmäßigen Treffen, in denen abgeschlossene Arbeitspakete dokumentiert und neue Zielvorgaben bis zum nächsten Treffen festgelegt wurden. Die Ergebnisse wurden jeweils in Protokollen festgehalten. Eine detaillierte Auflistung der Tätigkeiten sowie deren Zuordnung zu den einzelnen Teammitgliedern findet sich im ?? ??.

4.2 Anforderungen

Die Anforderungen an das Projekt wurden in Zusammenarbeit mit dem betreuenden Professor sowie dem Laboringenieur des MCT-Labors Hr. Lenkowski nach einer ersten Machbarkeitsanalyse im Hinblick auf die definierten Projektziele festgelegt. Neben den obligatorischen Grundfunktionen eines digitalen Speicheroszilloskops (siehe ?? “??“) wurden folgende zentrale technische Parameter bestimmt:

- Bandbreite: 1 MHz (Grenzfrequenz des Anti-Aliasing-Filters)
- Abtastrate: mindestens 10 MS/s (siehe ??)
- Auflösung: mindestens 10 Bit
- Analoge Eingangsspannung: ± 5 V
- Offset: ± 5 V (bezogen auf die Eingangsspannung)

Ein weiterer wesentlicher Aspekt war die *Spannungsversorgung* des Systems, für die bewusst keine strikten Anforderungen formuliert wurden. Auf diese Weise sollte während des Entwicklungsprozesses ein höheres Maß an Flexibilität gewährleistet werden, um den Fokus auf die grundlegende Funktionalität des Gesamtsystems legen zu können. Die Signalaufnahme sollte auf *einen Kanal* beschränkt bleiben, um Schwierigkeiten bei der Synchronisation mehrerer ADCs sowie den erhöhten Verwaltungsaufwand für Mehrkanalsysteme zu vermeiden. Außerdem sollte die Möglichkeit bestehen, dass Oszilloskop mit einem Standardtastkopf zu betreiben. Zusätzlich war vorgesehen, die Baugruppe als *erweiterbares Steckboard für ein Mikrocontroller-Entwicklungsboard* umzusetzen, wodurch die Komplexität reduziert und die Funktionalität der Baugruppe in den Vordergrund gestellt werden konnte.

Für die Schnittstelle zwischen ADC und Mikrocontroller wurde eine *parallele Anbindung* vorgesehen, da eine serielle Übertragung über SPI bei den geforderten Abtastraten eine zu hohe Taktfrequenz erfordert hätte. Dies hätte einen direkten Datentransfer in den Speicher des Mikrocontrollers nicht mehr zuverlässig ermöglicht. Für die zentrale Recheneinheit wurde sich auf die *Mikrocontroller der STM32-Familie* beschränkt, um vorhandene Kenntnisse aus den Modulen Mikrocomputertechnik und Embedded Systems gezielt einzusetzen zu können. Die Kommunikation mit dem PC sollte über eine *USB-Schnittstelle* erfolgen, um eine möglichst uneingeschränkte Kompatibilität zu gewährleisten. Auf dem PC war die Entwicklung einer *eigenen Anwendung zur Visualisierung und Verarbeitung*

der Messdaten vorgesehen. Diese sollte über *genormte Kommandos* kommunizieren, sodass auch eine Integration in gängige Entwicklungs- und Analyseumgebungen wie *LabVIEW* oder *MATLAB* möglich wäre.

Neben diesen zentralen Vorgaben wurden auch *optionale Erweiterungen* definiert, die nicht als kritisch für den Projekterfolg eingestuft wurden oder den vorgesehenen Arbeitsumfang überschritten. Dazu zählten insbesondere die Implementierung eines *Logikanalysators* mit Protokolldekodierung gängiger Schnittstellen (z. B. SPI, I²C, UART), die Realisierung einer *Fast-Fourier-Transformation (FFT)* zur Spektrumanalyse, die Erweiterung um einen *zweiten analogen Kanal*, die Integration von *Mathematik- und Messfunktionen* sowie die Nutzung unterschiedlicher *Triggerbedingungen*. Weitere Optionen im Hardwarebereich betraten eine *Optimierung hinsichtlich elektromagnetischer Verträglichkeit (EMV)*, eine *Minimierung des Rauschverhaltens*, die *Stromversorgung über USB* sowie die *Integration des Mikrocontrollers direkt auf der Baugruppe*.

4.3 Konzept / Architektur

Parallel zu den Anforderungen wurde ein Konzept entwickelt, um die grundlegende Architektur und Aufgabenverteilung des Projekts festzulegen. Zu Beginn entstand ein grundlegendes, rein funktionales Übersichtsbild (siehe ??). Dieses stellt die wesentlichen Signalflüsse und Verarbeitungsschritte dar, die für die Realisierung des USB-Oszilloskops erforderlich sind. Der Signalpfad führt dabei vom Device under Test (DUT) über die Signalerfassung zur Speicherung der Rohdaten und weiter zur Verarbeitung auf PC-Ebene. Dieses Schema diente als Orientierungshilfe für die weiteren konzeptionellen Schritte, ohne jedoch bereits konkrete Funktionseinheiten Bauteile oder Schnittstellen festzulegen.

Auf Basis dieser funktionalen Skizze sowie des in der Fachliteratur beschriebenen Grundaufbaus eines digitalen Speicheroszilloskops (siehe ??) wurde schließlich ein Gesamtkonzept entwickelt, das Funktionseinheiten definiert und den jeweiligen Bereichen Hardware, Firmware und Software zuordnet (siehe ??).

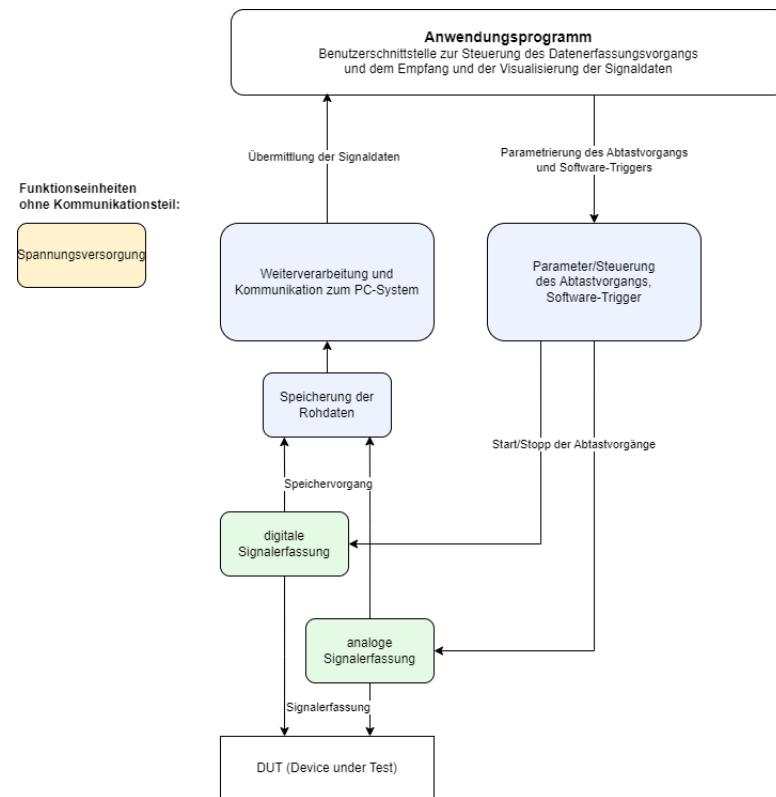


Abbildung 21: Übersichts-Blockschaltbild (erstellt für die Konzeptvorstellung)

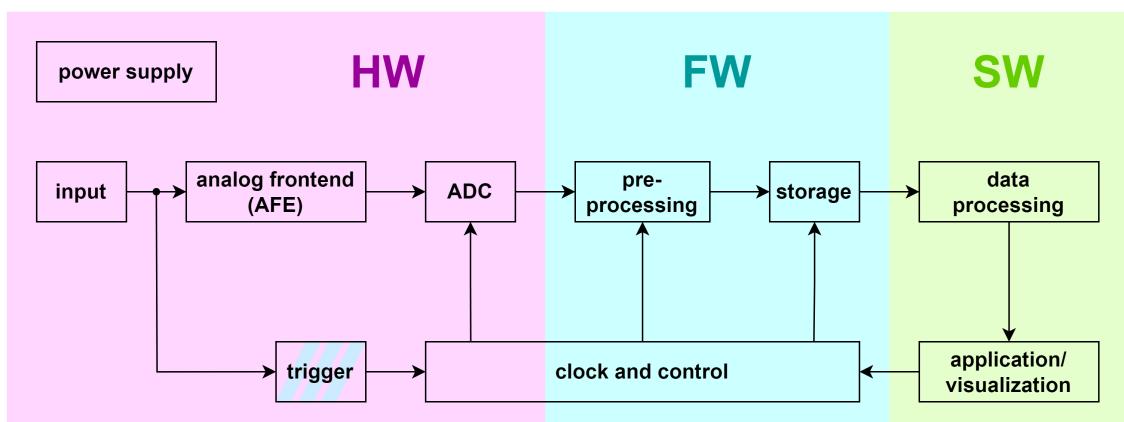


Abbildung 22: Gesamtkonzept des USB-Oszilloskops (nach [Muehl2020])

Ausgehend von diesem Konzept und der klaren Aufteilung in die Bereiche Hardware, Firmware und Software, wurden die entsprechenden Schnittstellen festgelegt und die Aufgabenbereiche verteilt.

5 Hardware (HW)

5.1 Entwurf

5.2 Implementierung

5.3 HW-Test

6 Schnittstelle Hardware - Firmware

7 Firmware (FW)

7.1 Entwurf

7.1.1 Auswahl des Mikrocontrollers

Eine zentrale Entscheidung im Entwurf der Firmware war die Auswahl eines geeigneten Mikrocontrollers (μ C). Dieser stellt den zentralen Knotenpunkt der Gesamtanwendung dar. Hierzu wurden nach der Erstellung des Gesamtkonzepts zunächst die Anforderungen definiert, die an den Mikrocontroller gestellt werden. Bei der Auswahl musste berücksichtigt werden, dass ein *Direct Memory Access (DMA)* vorhanden ist. Dieser ermöglicht die Speicherung der Daten nach Eintreten eines Trigger-Ereignisses ohne Umweg über die CPU. Außerdem wurde gemäß der Anforderungen im Gesamtkonzept eine *USB-Schnittstelle* benötigt, um die Kommunikation des Mikrocontrollers (USB-Device) mit dem PC (USB-Host) sicherzustellen. Dabei war es von Vorteil, wenn der Controller mindestens Full-Speed-Übertragungen mit 12 Mbit/s, idealerweise High-Speed-Übertragungen mit 480 Mbit/s unterstützt. Weiterhin war es erforderlich, dass ein *erschwingliches Entwicklungsboard* verfügbar ist, welches für den ersten Prototypen genutzt werden kann. Dieses sollte zunächst zum Einsatz kommen, bevor der Mikrocontroller zusammen mit den übrigen Schaltungsteilen auf eine gemeinsame Leiterplatte integriert wird. Darüber hinaus sollten *standardisierte Schnittstellen* wie I²C und SPI vorhanden sein, um die Anbindung eines externen ADCs auch auf andere Weise zu ermöglichen, falls die parallele Schnittstelle nicht umsetzbar wäre. Ein *hoher GPIO Input Speed* war notwendig, um die geplante parallele Anbindung eines externen ADCs mit 12 Bit Auflösung über zwölf parallele GPIO-Leitungen bei der geplanten Abtastrate zu gewährleisten. Außerdem musste der Mikrocontroller über *ausreichend großen SRAM-Speicher* verfügen, um die erfassten Abtastwerte ablegen zu können.

$$[\text{Benötigter Speicher}] = [\text{Anzahl Abtastwerte}] \cdot \left[\frac{2\text{Byte}}{\text{Abtastwert}} \right] \quad (14)$$

$$[\text{Anzahl Abtastwerte}] = \frac{[\text{gewünschte Abtastzeit}]}{[\text{Abtastperiode}]} \quad (15)$$

Die Anforderungen wurde unter der Annahme formuliert, dass ein externer ADC eingesetzt werden sollte. Die angestrebte Abtastrate von etwa 10 MHz musste auch mikrocontrollerseitig erreichbar sein.

Auf Grundlage der positiven Erfahrungen mit STM32-Mikrocontrollern der Firma STMicroelectronics sowie der integrierten Entwicklungsumgebung STM32CubeIDE im MCT-Praktikum, wurde entschieden, einen Mikrocontroller aus dieser Serie auszuwählen. Ein zusätzlicher Vorteil dieser Produktfamilie ist die Verfügbarkeit kostengünstiger Entwicklungsboards (NUCLEO-Boards), die sich hervorragend für die ersten Entwicklungsschritte eignen und die Materialkosten des Projekts reduzieren.

Nach einem Vergleich verschiedener STM32-Produktfamilien in Bezug auf die definierten Anforderungen fiel die Wahl auf die [STM32F7-Serie](#). Diese Mikrocontroller basieren auf einem ARM Cortex-M7 und gehören zum High-Performance-Bereich der STM32-Serie. Lediglich die STM32H7-Serie bietet noch höhere Leistung, da sie über einen weiteren Prozessorkern (Cortex-M4-Core) verfügen. Schlussendlich wurde der [STM32F767ZI](#) als Mikrocontroller ausgewählt. Dieser bietet im Vergleich zu anderen Controllern derselben Familie einen größeren Flash- und SRAM-Speicher. Zudem ist ein passendes Entwicklungsboard, das [NUCLEO-F767ZI](#), verfügbar. Ein weiterer Vorteil dieses Boards ist das integrierte Programmiergerät, das durch einen separaten Mikrocontroller mit entsprechender Firmware realisiert wird.

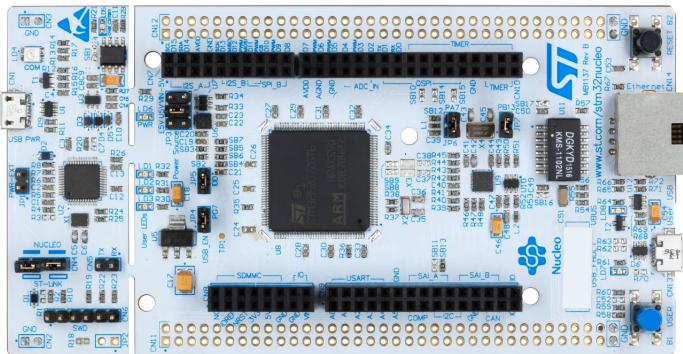


Abbildung 23: Beispielabbildung für ein NUCLEO-144-Board, wie das NUCLEO-F767ZI
(Quelle: [\[Link\]](#))

Nachdem die Entwicklungsboards in Absprache mit dem MCT-Labor beschafft worden waren, erfolgte eine erneute Einarbeitung in die Arbeit mit ARM Cortex-M Mikrocontrollern und den spezifischen Funktionen der STM32-Hardware.

7.1.2 Konzept des Abtastprozesses

Der mikrocontrollerseitige Abtastprozess gliedert sich in zwei Hauptaufgaben:

- Das *Feststellen des Aufzeichnungsbeginns* durch ein definiertes Triggerereignis und
- die *Aufnahme und Speicherung* der digitalisierten Abtastwerte.

Das Konzept der Abtastung ist als Funktionsschema in ?? dargestellt.

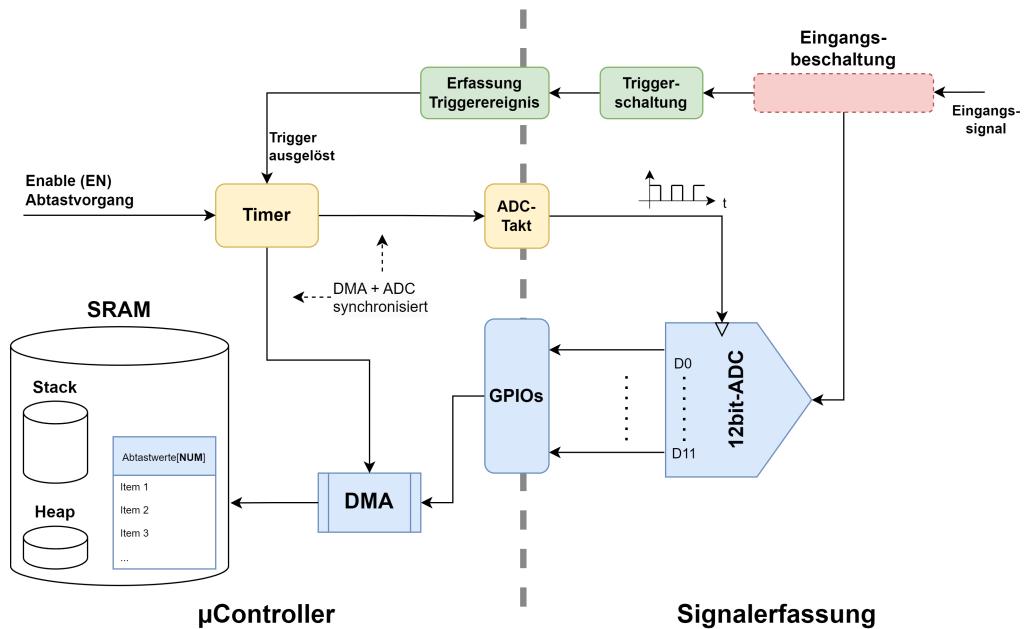


Abbildung 24: Funktionsschema der Abtastung

Parallele Datenanbindung

Für die Übertragung der Messdaten vom externen ADC zum Mikrocontroller wurde eine *parallele Schnittstelle* umgesetzt. Jedes der zwölf Bits des 12-Bit-ADCs wird dabei über eine eigene Datenleitung an den Mikrocontroller herangeführt. Der wesentliche Vorteil dieser Anbindung liegt in der vergleichsweise einfachen Implementierung, da kein komplexes serielles Übertragungsprotokoll notwendig ist. Ein Nachteil besteht jedoch darin, dass bei hohen Abtastraten Synchronisationsprobleme auftreten können.

Die GPIO-Pins des Mikrocontrollers sind in Ports mit jeweils 16 Pins organisiert, die sich ein gemeinsames *Input Data Register (IDR)* teilen. Da für die Parallelschnittstel-

le zwölf Leitungen benötigt werden, können alle Bits der ADC-Ausgabe in einem einzigen Lesezugriff erfasst werden. Die Aktualisierung des IDR erfolgt mit jedem Zyklus des AHB-Busses [STmicroelectronics2024], der mit bis zu 216 MHz betrieben werden kann (siehe Clock-Tree in [STmicroelectronics2024]). Dieser Wert wird aber durch eine vorhandene Eingangskapazität der GPIOs reduziert, welche eventuell auf einen neuen Wert (Spannungsspeigel) umgeladen werden müssen. Die tatsächliche, maximal mögliche Abtastfrequenz hängt also von dieser Eingangskapazität und der treibenden Schaltung ab. Im Datenblatt des μ C ist für die Eingangskapazität ein Wert von $C_{IO} = 5pF$ angegeben ([STmicroelectronics2025]). Maximale Änderungsfrequenzen sind aber nur für die Nutzung der GPIOs als Output angegeben (die treibende Stufe ist dann die Ausgangsstufe des GPIO). Hier werden, je nach Einstellung des GPIO-Speeds (konfigurierbar), selbst im langsamsten Modus noch Frequenzen im einstelligen MHz-Bereich erreicht ([STmicroelectronics2025]). Unter der Annahme, dass die Treiberfähigkeit der ADC-Datenausgänge höher ist, kann die Abtastrate von 10 MHz realisiert werden.

Speicherorganisation

Die erfassten Abtastwerte werden zunächst im internen SRAM des Mikrocontrollers zwischengespeichert. Der STM32F767 verfügt über insgesamt 512 KB SRAM, aufgeteilt in mehrere Speicherblöcke [STmicroelectronics2024] (Memory Map). Für das Projekt wurde zunächst der *SRAM1-Bereich* (368 KB) für die Speicherung der Messdaten reserviert, während der *DTCM-Bereich* (128 KB) für Stack und Heap vorgesehen wurde. Diese Aufteilung gewährleistet eine klare Trennung zwischen Programmdaten und Messwertspeicher und kann bei der Implementierung im Linker-Script festgelegt werden.

DMA-basierter Datentransfer

Ein direkter Zugriff der CPU auf die GPIO-Datenregister und das anschließende Abspeichern der Daten in den SRAM würde den Prozessor vollständig auslasten und keine parallele Ausführung anderer Aufgaben ermöglichen. Aus diesem Grund wurde der Datentransfer mithilfe des Direct Memory Access (DMA) realisiert (siehe ??).

Beim gewählten Ansatz wurde der DMA-Controller so konfiguriert, dass er den Wert des Eingangsdatenregisters der GPIOs automatisch in den SRAM schreibt, ohne dass die CPU beteiligt ist (siehe Schema in ??). Jeder DMA-Transfer wird durch einen Hardware-

Timer (TIM1) angestoßen. Dadurch kann eine exakte Synchronisation zwischen dem Clock-Signal des ADC und der Datenübertragung erreicht werden. Die Quelladresse bleibt während der gesamten Aufzeichnung konstant (Adresse des IDR), während die Zieladresse nach jedem Transfer um zwei Byte inkrementiert wird, um den nächsten Speicherplatz im SRAM anzusprechen.

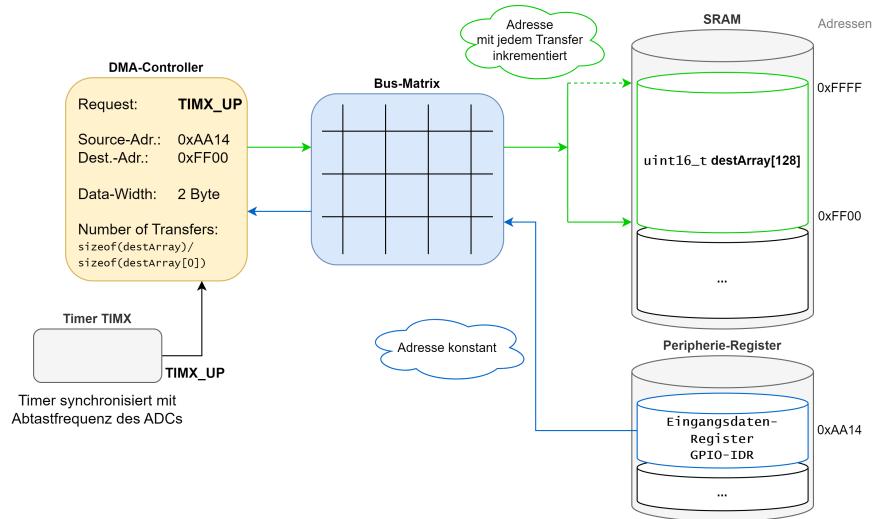


Abbildung 25: Funktionsprinzip DMA Abtastung - Adresswerte dienen nur der Übersicht

Da das NDTR-Register (Number of Data Transfers Register) maximal 65.535 Transfers speichern kann [STmicroelectronics2024], wird der Double-Buffer Mode (DBM) des DMA-Controllers eingesetzt (benötigt werden z.B. 100.000 Transfers für 100.000 Abtastwerte). In diesem Modus werden abwechselnd zwei getrennte Zieladress-Zeiger genutzt, um kontinuierliche Transfers ohne Datenverlust zu ermöglichen. Sobald ein Speicherbereich befüllt ist, wird automatisch in den anderen Bereich weitergeschrieben. Gleichzeitig kann die Software den gefüllten Speicherbereich auswerten oder für den USB-Transfer vorbereiten, ohne den laufenden Datentransfer zu unterbrechen.

Trigger-Mechanismus

Der Beginn der Aufzeichnung wird durch ein definiertes Triggerereignis gesteuert. Nach dem Auslösen dieses Ereignisses startet der DMA automatisch mit dem Füllen des vorsehenen Speicherbereichs. Die Größe dieses Bereichs sowie die Anzahl der Transfers wurde zuvor in der Firmware festgelegt. Sobald der Speicher vollständig gefüllt ist, wird der Ab-

tastprozess automatisch beendet, und die Daten stehen für die anschließende Übertragung an den PC bereit. Der Trigger-Mechanismus wurde zunächst mit Hardware-Interrupts geplant (bei STM32: Extended Interrupts - EXTI). Mit dem EXTI-Modul kann ein Aufruf eines Exception-Handlers³ bei einer steigenden oder fallenden Flanke an einem GPIO erfolgen, in welchem der DMA für die Datentransfers aktiviert wird⁴.

Synchronisation mit ADC-Takt

Für eine störungsfreie Datenaufnahme muss der Datentransfer exakt auf den ADC-Takt abgestimmt werden. Hierfür wurde das Taktsignal für den ADC durch denselben Timer erzeugt, der auch die DMA-Requests auslöste. Die Hardware-Timer der STM32-Controller besitzen sogenannte Capture-Compare-Einheiten mit sogenannten Output-Compare-Anschlüssen (OC), welche mit bestimmten GPIOs des μ C verbunden werden können (siehe [STmicroelectronics2024]). Der Hardware-Timer kann hierdurch einen GPIO ohne CPU-Beteiligung ansteuern. Um ein Setup-Violation-Problem zu vermeiden, wurde der Takt invertiert: Die positive Flanke des Signals erzeugt am ADC-Ausgang einen neuen Wert, während synchron zur negativen Flanke der DMA-Transfer ausgelöst wird. Dadurch können sich die ADC-Daten für eine halbe Abtastperiode $\frac{T_S}{2}$ stabilisieren, bevor sie eingelesen werden. Dies stellt eine zuverlässige Digitalisierung sicher (siehe ??).

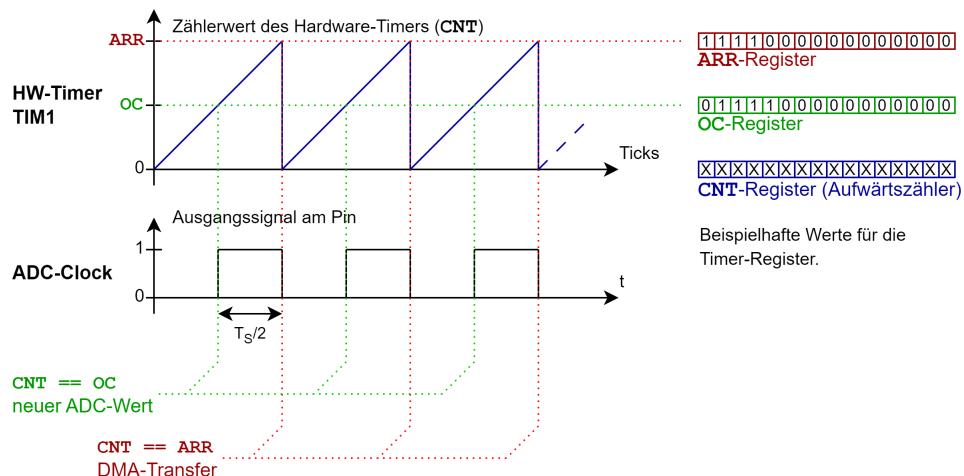


Abbildung 26: Zeitdiagramm und Registerübersicht des Hardware-Timers für die Synchronisation von ADC und DMA (Sampling am μ C)

³Bezeichnung für eine Interrupt-Service-Routine (ISR) bei ARM-Cortex- μ C.

⁴Die Zeit zwischen der Triggerflanke und DMA-Aktivierung im Exception-Handler ist die Triggerlatenz.

7.1.3 Konzept der Firmware als FSM

Überblick

Die Firmware des USB-Oszilloskops wurde auf der Basis einer Finite State Machine (FSM) zur Steuerung der Betriebszustände erstellt. Dieser Ansatz ermöglicht eine klare Trennung der verschiedenen Funktionsphasen sowie eine strukturierte und erweiterbare Implementierung. Die FSM verarbeitet asynchrone Ereignisse (Events; Präfix EV_), die aus unterschiedlichen Quellen stammen können, wie z.B.:

- Benutzereingaben (z. B. Button auf dem Board),
- Befehle der PC-Anwendung über USB,
- interne Signale der Hardware, wie z. B. Abschluss eines DMA-Transfers.

Die fachlichen Grundlagen zu FSMs wurden bereits in ?? erläutert.

Um diese Ereignisse effizient zu verarbeiten, wird eine Event-Queue verwendet. Jedes Event wird in diese Warteschlange (Queue) eingereiht („enqueued“). Die FSM verarbeitet die Ereignisse dann sequenziell, wodurch sichergestellt wird, dass auf asynchrone Signale deterministisch reagiert wird.

Architektur der FSM

Die FSM besteht aus mehreren definierten Zuständen, die jeweils einer bestimmten Betriebsphase des Oszilloskops entsprechen. Jeder Zustand wird durch eine eigene Funktion realisiert. Diese Funktionen beinhalten drei Hauptbereiche:

1. *Entry-Block*:

Wird beim Eintritt in einen Zustand ausgeführt (z. B. Initialisierungen, Statusmeldungen).

2. *Event-Handling*:

Reaktion auf ankommende Events innerhalb des aktuellen Zustands.

3. *Exit-Block*:

Wird beim Verlassen des Zustands ausgeführt (z. B. Aufräumarbeiten).

Das State Diagram (siehe ??) visualisiert die Abfolge der Firmwarezustände sowie die möglichen Übergänge basierend auf bestimmten Events. Die Hauptzustände sind ?? zu entnehmen.

Zustand	Beschreibung
Init	Initialisierung der Hardware-Peripherie. Wird nur einmal beim Start ausgeführt.
Idle	Warten auf Konfigurationsbefehle oder Startsignal der PC-Anwendung.
Acquisition Start	Aktivieren des Triggers und Starten der Datenerfassung.
Acquisition Done	Abschluss der Datenerfassung, Vorbereitung der Daten für die Übertragung.
Data Transmission	Übertragung der erfassten Daten an den PC.
Wait for ACK	Warten auf Bestätigung vom PC, ob ein weiterer Erfassungsvorgang gestartet werden soll.

Tabelle 1: Übersicht über die Hauptzustände der Firmware-FSM

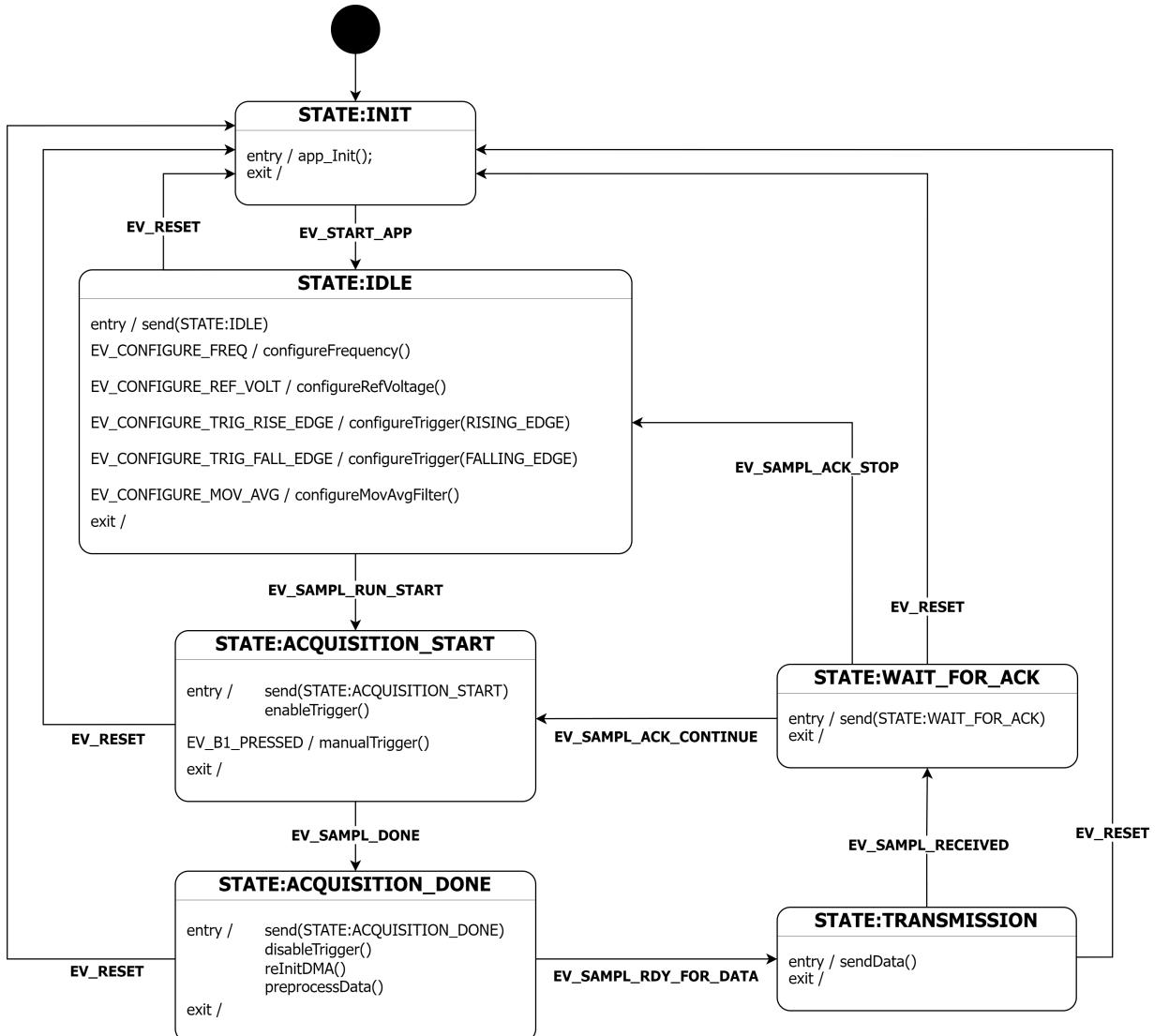


Abbildung 27: Zustandsdiagramm der Firmware

Event-Queue

Die FSM verarbeitet Ereignisse über eine zentrale Event-Queue. Dies erlaubt eine entkoppelte Kommunikation zwischen verschiedenen Modulen der Firmware und stellt sicher, dass Ereignisse nicht verloren gehen, auch wenn mehrere nahezu gleichzeitig auftreten.

Prinzipieller Ablauf:

1. Ein Modul erkennt ein Ereignis (z. B. Button-Druck, DMA-Abschluss, USB-Befehl).
2. Das Ereignis wird als Event-Code in die Queue geschrieben („enqueued“).
3. Die Hauptschleife der Firmware liest fortlaufend Events aus der Queue („dequeued“) und leitet sie an die FSM weiter.
4. Die FSM entscheidet anhand des aktuellen Zustands und des Event-Codes, wie reagiert wird und ob ein Zustandswechsel nötig ist.

Vorteile dieses Ansatzes:

- Asynchrones Verhalten wird klar handhabbar.
- Saubere Trennung zwischen Ereigniserzeugung und -verarbeitung.
- Erweiterbar für zukünftige Ereignisse, ohne die Hauptarchitektur zu ändern.

Zustandsübergänge

Die wichtigsten Zustandsübergänge lassen sich wie folgt zusammenfassen:

- *Systemstart:*
Init → Idle nach erfolgreicher Initialisierung.
- *Start der Datenerfassung:*
Idle → Acquisition Start nach Empfang des Startbefehls von der PC-Anwendung.
- *Abschluss der Datenerfassung:*
Acquisition Start → Acquisition Done nach Abschluss der DMA-Operation.
- *Übertragung der Daten:*
Acquisition Done → Data Transmission sobald PC signalisiert, dass er bereit ist.
- *ACK-Verarbeitung:*
 - Wait for ACK → Acquisition Start für weitere Messung.
 - Wait for ACK → Idle für Beendigung der Erfassung.

7.1.4 Preprocessing-Konzept

Umrechnung 12bit signed (ADC) in 16bit signed

Die Übertragung der Abtastwerte erfolgt in Einheiten von Halfwords (2 Byte). Da die Rohdaten des ADCs als 12-Bit signed Integer vorliegen, werden sie vor der Übertragung in einen gängigen Datentyp umgerechnet. Verwendet wird dabei ein 16-Bit signed Integer (`int16_t` in C), da ein nativer 12-Bit Integer-Datentyp nicht existiert.

Durch diese Umwandlung können die Daten auf der PC-Seite direkt weiterverarbeitet werden, ohne dass zusätzliche komplexe Konvertierungsschritte erforderlich sind. Insbesondere kann die Empfangssoftware (Python-Anwendung) die empfangenen Werte unmittelbar als 16-Bit signed Integer interpretieren und anschließend in den gewünschten Zieldatentyp umwandeln. Ein weiterer Vorteil der Umrechnung liegt in der verbesserten internen Weiterverarbeitung auf Mikrocontroller-Seite. So konnten die Daten vor dem Versand beispielsweise effizient für die Anwendung eines gleitenden Mittelwertfilters aufbereitet werden (vgl. nächster Abschnitt).

Nachfolgend sind mögliche Algorithmen als Python-Code beschrieben:

1. Sign-Extend (Shift-Methode)

```
1 def sign_extend_12_to_16(value12: int) -> int:
2     # 12-Bit Zahl um 4 nach links, dann arithmetisch nach rechts
3     return (value12 << 4) >> 4
```

Listing 1: Sign-extend (Arithmetisches Shift schiebt das MSB nach)

2. Zweierkomplement rückwärts: invertieren → +1

```
1 def twos_complement_backward_invert_plus1(value12: int) -> int:
2     if value12 & 0x800: # Vorzeichenbit prüfen
3         inverted = (~value12) & 0xFFFF # Nur 12 Bits
4         magnitude = inverted + 1
5         return -magnitude
6     else:
7         return value12
```

Listing 2: Zweierkomplement rückwärts aufgelöst.

3. Zweierkomplement rückwärts: $-1 \rightarrow$ invertieren

```
1 def twos_complement_backward_minus1_invert(value12: int) -> int:
2     if value12 & 0x800: # Vorzeichenbit pruefen
3         decreased = (value12 - 1) & 0xFFFF
4         magnitude = (~decreased) & 0xFFFF
5         return -magnitude
6     else:
7         return value12
```

Listing 3: Alternative zu 2.: Erst $-1 \rightarrow$ dann invertieren.

Gleitender Mittelwertfilter

In ?? wurden bereits die Grundlagen zur digitalen Filterung von Signalen geliefert. Im Projekt wurde die DSP mittels *gleitendem Mittelwertfilter* umgesetzt, welcher zufälliges Rauschen aus dem Signal herausfiltern kann. Dies ist insbesondere bei verrauschten Signalquellen mit kleiner Ausgangsamplitude relevant. In ?? ist ein Filteralgorithmus beschrieben, der einen kumulativen gleitenden Mittelwertfilter umsetzt (vgl. [Smith1999]). Die Grundlagen des Filters wurden ebenfalls bereits in ?? erläutert.

```
100 'MOVING AVERAGE FILTER
110 'This program filters 5000 samples with a 101 point moving
120 'average filter, resulting in 4900 samples of filtered data.
130 '
140 DIM X[4999]           'X[ ] holds the input signal
150 DIM Y[4999]           'Y[ ] holds the output signal
160 '
170 GOSUB XXXX            'Mythical subroutine to load X[ ]
180 '
190 FOR I% = 50 TO 4949    'Loop for each point in the output signal
200   Y[I%] = 0             'Zero, so it can be used as an accumulator
210   FOR J% = -50 TO 50    'Calculate the summation
220     Y[I%] = Y[I%] + X(I%+J%)
230   NEXT J%
240   Y[I%] = Y[I%]/101      'Complete the average by dividing
250 NEXT I%
260 '
270 END
```

Abbildung 28: Pseudocode: Moving-Average-Filter aus [Smith1999]

Dieser Pseudocode-Ausschnitt war die Basis für die Implementierung des gleitenden Mittelwertfilters.

7.2 Implementierung

7.3 FW-Test

8 Schnittstelle Firmware - Software

8.1 Konzept der Kommunikation (USB CDC)

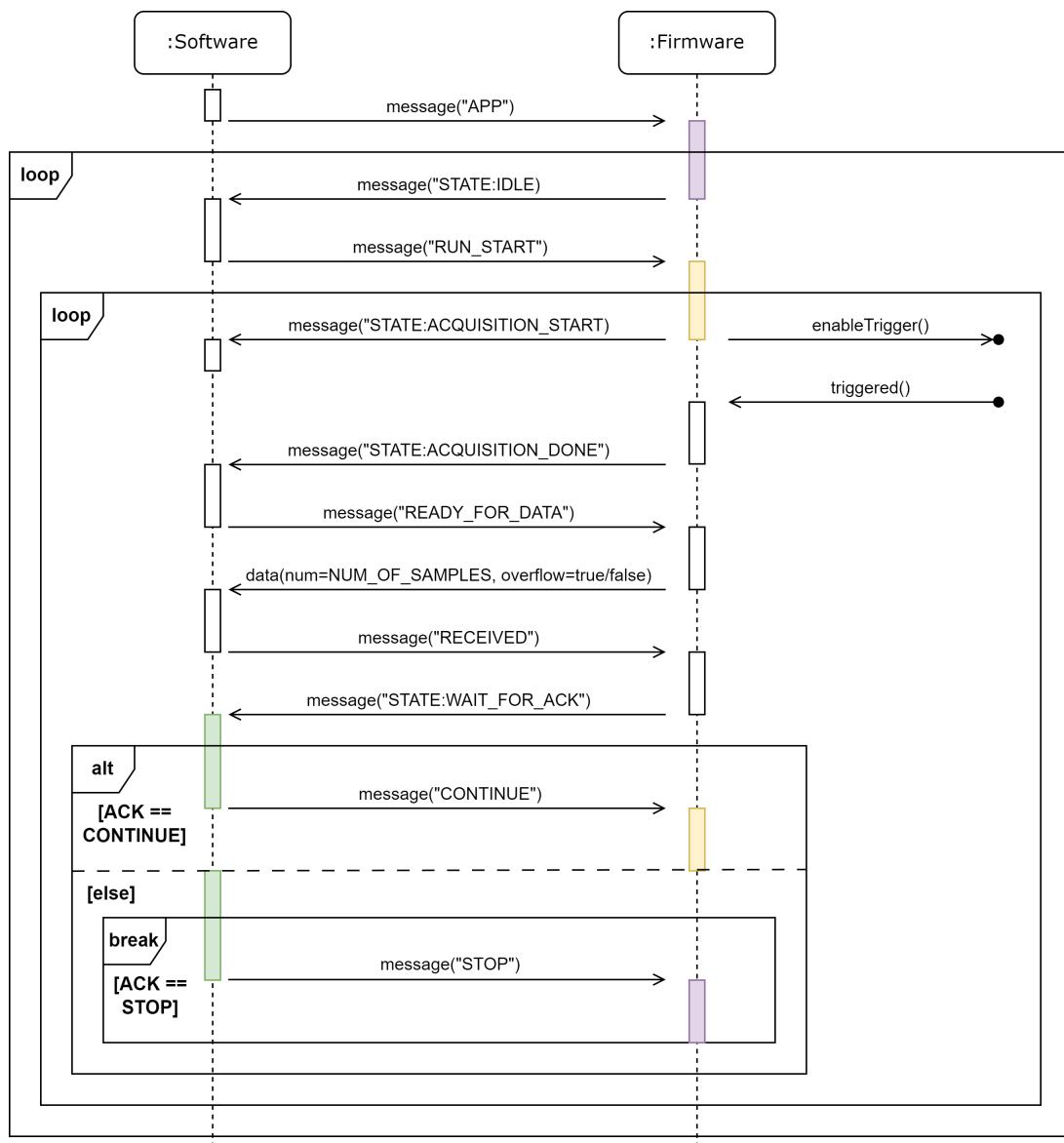


Abbildung 29: UML-Sequenzdiagramm: Kommunikation ohne Konfiguration

9 Software (SW)

9.1 Entwurf

9.2 Implementierung

9.3 SW-Test

10 Zusammenföhrung

10.1 Entwurf

10.2 Implementierung

10.3 SW-Test

11 Ergebnisse

12 Fazit und Ausblick

13 Literaturverzeichnis

14 Abbildungsverzeichnis

Abbildungsverzeichnis

A Anhang