Projekt USB-Oszilloskop

Samuel Oeser, Nicole Sturm, Daniel Wirth

12. 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 (μC) 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

Mühl: [**Muehl2020**]

3.2 Leitungsimpedanzanpassung

3.3 ADC-Topologien

3.4 AAF-Entwurf (Nyquisttheorem)

3.5 Frequenzkompensierter Spannungsteiler

Schrüfer: [**Schruefer2022**]

3.6 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.

3.6.1 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 $\delta : S \times E \to S$,
- einer Ausgabefunktion $\lambda : S \times E \to A$,
- 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.

3.6.2 Modellierung und Darstellung

Die Darstellung eines Zustandsautomaten erfolgt üblicherweise mit einem *Zustandsdiagramm* (auch Zustandsübergangsdiagramm), einem gerichteten Graph[1]. 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) (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]
[2] aktueller UML-Standard (V2.5.1): [Link]

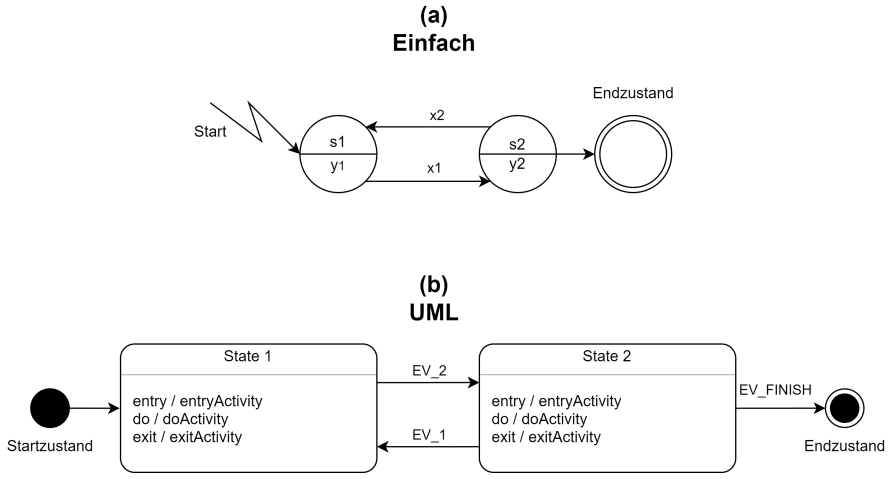

Abbildung 1: (a) Zustandsdiagramm (nach [**Baesig2019**]); (b) UML-Zustandsdiagramm

3.6.3 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-Programmieren genutzt werden.
- *Verbessertes Debugging*
  Da Zustandsübergänge zentralisiert sind, lassen sich Log-Messages und weitere Debugging-Mechanismen leichter verwenden.

3.7 Direct Memory Access - DMA

3.7.1 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 µC, die solche Transfers durchführt, heißt *Direct Memory Access Controller (DMAC)* (vgl. [**Urbanek2020**]). Im Folgenden soll die Funktionsweise anhand der Realisierung in der im Projekt verwendeten STM32-Mikrocontrollerfamilie erläutert werden.

3.7.2 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 Mikrocontrollers (µCs) 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.

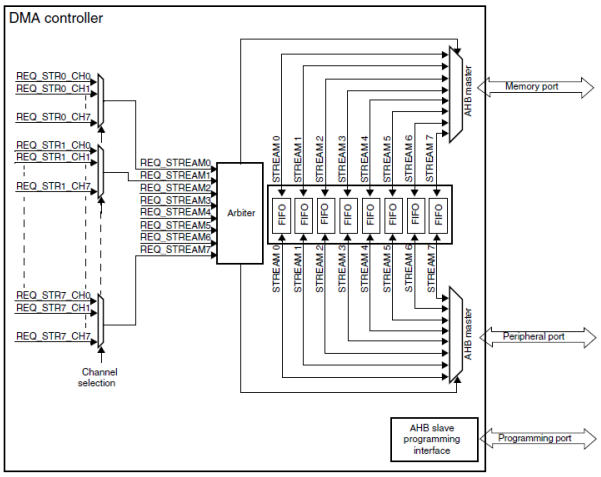

Abbildung 2: Blockdiagramm STM32-DMA-Controller aus [**STmicroelectronics2016**]

Der µC besitzt zwei unabhängige DMAC, 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*: peripherieseitiger Datenanschluss
  - *memory port*: speicherseitiger Datenanschluss

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 *Channeln*, 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 **??**).

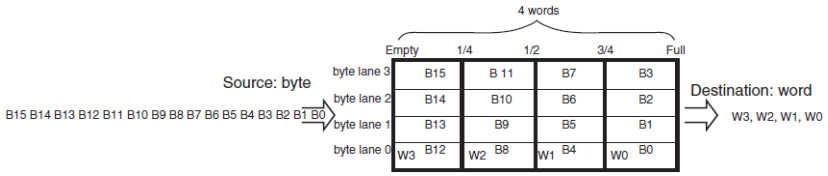

Abbildung 3: FIFO-Struktur aus [**STmicroelectronics2016**] (Teilabbildung)

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 Zeiladresse können jeweils aber auch konstant gehalten werden (siehe späteres Beispiel).

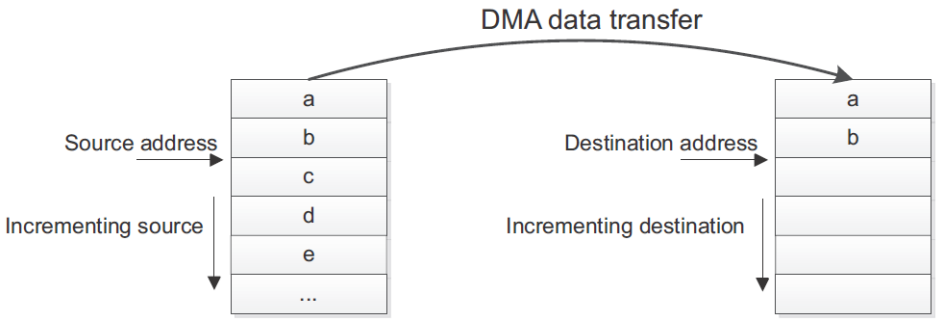

Abbildung 4: DMA Quell- und Zeiladressinkrementierung aus [**STmicroelectronics2016**]

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.

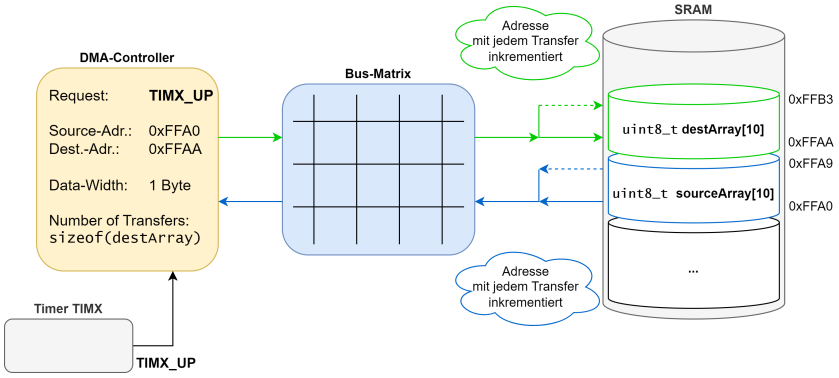

Abbildung 5: peripheral-to-memory-Transfer

Im Abtastsystem ist keine Inkrementierung der Quelladresse notwendig, da nur das Eingangsdatenregister ausgelesen werden muss, welches die Schnittstelle zum parallelen ADC-Interface darstellt.

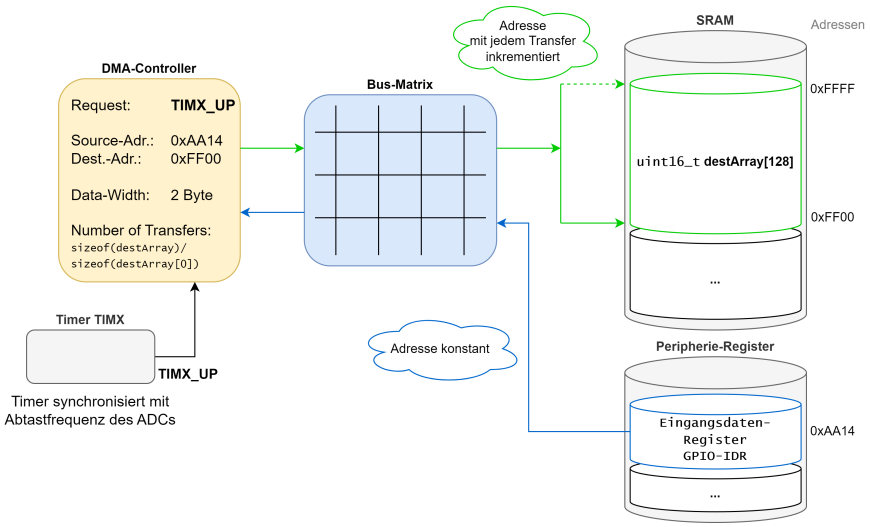

Abbildung 6: Funktionsprinzip DMA Abtastung

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.

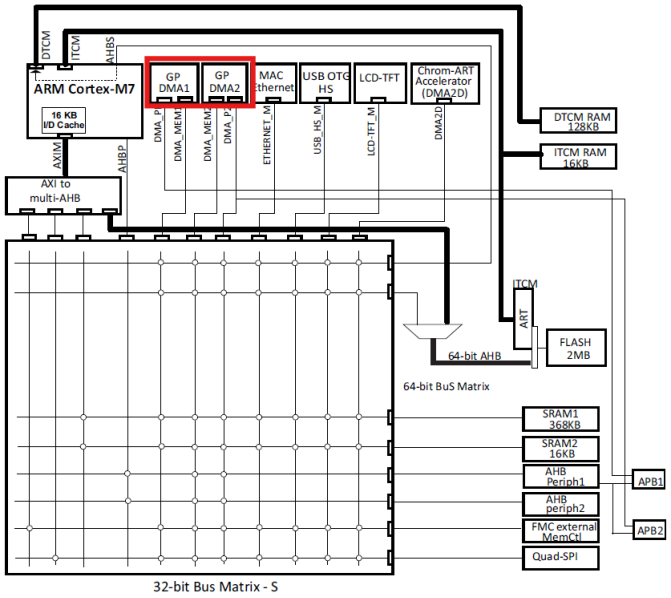

Abbildung 7: Systemarchitektur für STM32F76xxx μC aus [**STmicroelectronics2024**]

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

3.8 Digitale Filterung (Preprocessing)

The Scientist & Engineer's Guide to Digital Signal Processing, 1999

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.

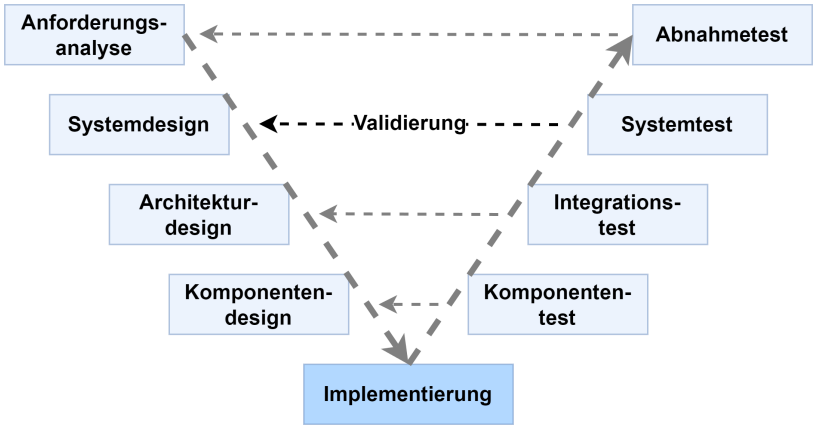

Abbildung 8: V-Modell

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 einsetzen 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 betrafen 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 **??**).

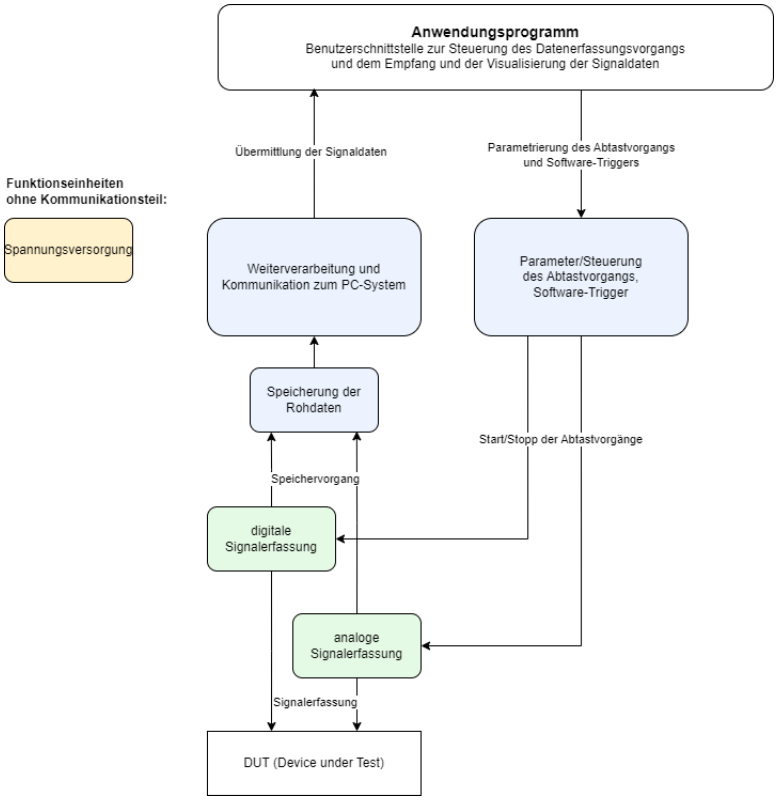

Abbildung 9: Übersichts-Blockschaltbild (erstellt für die Konzeptvorstellung)

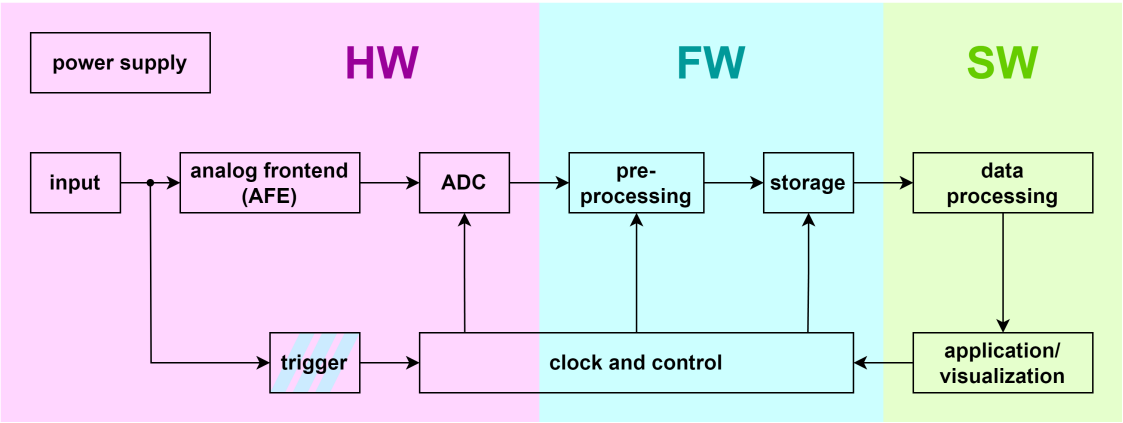

Abbildung 10: Gesamtkonzept des USB-Oszilloskops (nach [**Muehl2020**])

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 Realisierung

5.1 Hardware (HW)

5.1.1 Entwurf

5.1.2 Implementierung

5.1.3 HW-Test

5.2 Schnittstelle Hardware - Firmware

5.3 Firmware (FW)

5.3.1 Entwurf

5.3.2 Implementierung

5.3.3 FW-Test

5.4 Schnittstelle Firmware - Software

5.5 Software (SW)

5.6 Zusammenführung

5.6.1 Entwurf

5.6.2 Implementierung

5.6.3 SW-Test

6 Ergebnisse

7 Fazit und Ausblick

8 Literaturverzeichnis

9 Abbildungsverzeichnis

Abbildungsverzeichnis

A Anhang