Malte Seldschopf

Erstellung eines Phasor-Messgerätes

Praxisprojekt

Technische Hochschule Köln,

Cologne Institut for Renewable Energy,

07. 01. 2019

Betreuer: Prof. Dr. Eberhard Waffenschmidt

Cologne Institute for Renewable Energy

Technology Arts Sciences TH Köln

Kurzfassung

Das Praxisprojekt beschäftigt sich mit der Entwicklung eines Phasor-Messgerätes und dient als Vorarbeit zu der darauf aufbauenden Bachelor Arbeit. Es wird konkret auf die Programmierung sowie die Ermittlung der Anforderung an ein solches Gerät eingegangen.
In der Bachelorarbeit soll mit Hilfe des Phasor-Messgeräts die Phasenverschiebung zweier Spannungen gemessen und verglichen werden. Durch diese Messung sollen Leistungsverluste ermittelt werden, welche z.B. zwischen Verbraucher und Erzeuger auftauchen. Zur Erstellung wurde eine Entwicklungsplatine der Firma Digilent verwendet. Auf dieser Entwicklungsplatine befindet sich ein Mikrocontroller der Firma Microchip, der PIC32MZ2048EFG100.

Abstract

This practical work deals with the development of a Phasor measurement Unit and serves as a preliminary work for the following bachelor thesis. The programming and the determination of the specifications for such a device are described in detail. The Phasor Measurement Unit is used to measure and compare the phase shift of two voltages. With the help of this measurement, power losses are to be determined, which occur e.g. between consumer and generator. A development board from Digilent was used for this purpose. This development board contains a microcontroller from Microchip, the PIC32MZ2048EFG100.

Inhaltsverzeichnis

1. Einleitung

Ziel dieser Arbeit ist es mir, dem Studierenden, eine letzte Übung im Schreiben von Abschlussarbeiten zu geben. Gleichzeitig soll das selbstständige Erkennen und Lösen von Problemen trainiert werden. Das Themengebiet dieser Arbeit kam mir gelegen. Denn durch meine Arbeit als Werksstudent bei der Firma Gude Analog- und Digitalsysteme GmbH, in der ich Geräte zur Phasor-Messung gefertigt habe, kann ich einen Bezug zur Phasor-Messgeräten herstellen. Durch das Fach EEV (Elektrische Energieverteilung) habe ich einen Bezug zur Energieverlusten auf Leitungen herstellen können. Diese beiden Schwerpunkte zusammengefasst, habe die Weichen für mein Praxisprojekt gelegt.

Das erste Kapitel bezieht sich auf das Hintergrundwissen der verwendeten Komponenten, worauf in den weiteren Kapiteln häufig Bezug genommen wird. Es soll das Verständnis und den Zusammenhang der einzelnen Komponenten miteinander verständlicher machen.

Kapitel zwei beschäftigt sich mit der verwendeten Hardware. Es werden zuerst die allgemeinen Anwendungsbereiche der Entwicklungsplatine beschrieben. Anschließend werden die Spezifikationen des Prozessors und des GPS-Empfängers genannt. Die detaillierte Beschreibung der verwendeten Hardware würde den Umfang dieses Berichts überschreiten, deshalb werden die allgemeinen Eigenschaften oberflächlich erörtert und präziser die verwendeten Bausteine, welche für das Programm notwendig sind.

Es folgt das Kapitel über das eigentliche Programm. Hier wird Abschnitt für Abschnitt, parallel zum Programmcode die Ziele der jeweiligen Programmzeile erläutert und wiedergegeben. Bibliotheken, sowie der Code, welcher für die Verbindung der SD-Karten genutzt wird, werden nur kurz erläutert, da nur vereinzelte Funktionen daraus genutzt werden.

2. Stand der Technik

Phasor-Messgeräte sind Messgeräte für sinusförmige Signale, die deren Amplitude und Phasenlage messen. Sie werden in der Niederspannungstechnik und in Smart-Grids eingesetzt und messen in kurzen Zeitabständen den Zustand der Wechselspannung.
Heutzutage gibt es eine Menge an Geräten, welche zur Phasenwinkel Messungen in der Lage sind. Um eine Aussage über die Netzstabilität zu geben, ist es notwendig einen Referenzpunkt als Ausgangslage zu verwenden, um einen Bezug zu der Veränderung des Netzes feststellen zu können. Das hier entwickelte Gerät muss dementsprechend noch ein zweites Mal gefertigt werde, es wird dann mit der hier gefertigten Software versehen. Sobald diese zwei Geräte im Betrieb sind und sich an unterschiedlichen Stellen des zu messenden Netzes befinden, können Aussagen über die Netzstabilität mithilfe der Ermittlung des Phasenwinkels getroffen werden.
Die meisten Geräte werden zur Strommessung eingesetzt und geben außerdem, den aktuellen Phasenwinkel an. Die Geräte der Firma Gude, z.B. der Expert Power Control 1105, zeigen immer den aktuellen Phasenwinkel an [1].

Abbildung 1: Phasor-Messgerät der Firma GUDE – EPC 1105

Quelle: Gude Analog- und Digitalsysteme GmbH [1]

Das in diesem Projekt entwickelte Gerät unterscheidet sich geringfügig von anderen Phasor-Messgeräten. Der deutlichste Unterschied liegt darin, dass das hier entwickelte Gerät auf einer vorher festgelegten Uhrzeit gestartet werden kann und die Daten daraufhin abspeichert. Außerdem werden mehrere Spannungsperioden aufgenommen und nicht wie in den vorhandenen Geräten, nur der aktuelle Wert. Da es eine Vielzahl von unterschiedlichen Phasor-Messgeräten gibt, ist es schwer zu sagen, wo genau die Differenzen zu dem hier entwickelten Gerät liegen, da die Geräte auch untereinander voneinander abweichen. Der wichtigste Kernpunkt eines Phasor-Messgerätes liegt in der Messung des Phasenwinkels und der Zeitgenauen Synchronisation, alles andere sind zusätzliche Eigenschaften.

3. Hintergrund

3.1. Entwicklungsumgebung

Das Praxisprojekt wurde auf der Entwicklungsumgebung, MPLAB X IDE von Mikrochip in C unter Windows 10 geschrieben. Es ist ein Softwareprogramm, welches auf einem PC läuft, um Anwendungen für Mikrochip-Mikrocontroller und digitale Signalcontroller zu entwickeln.

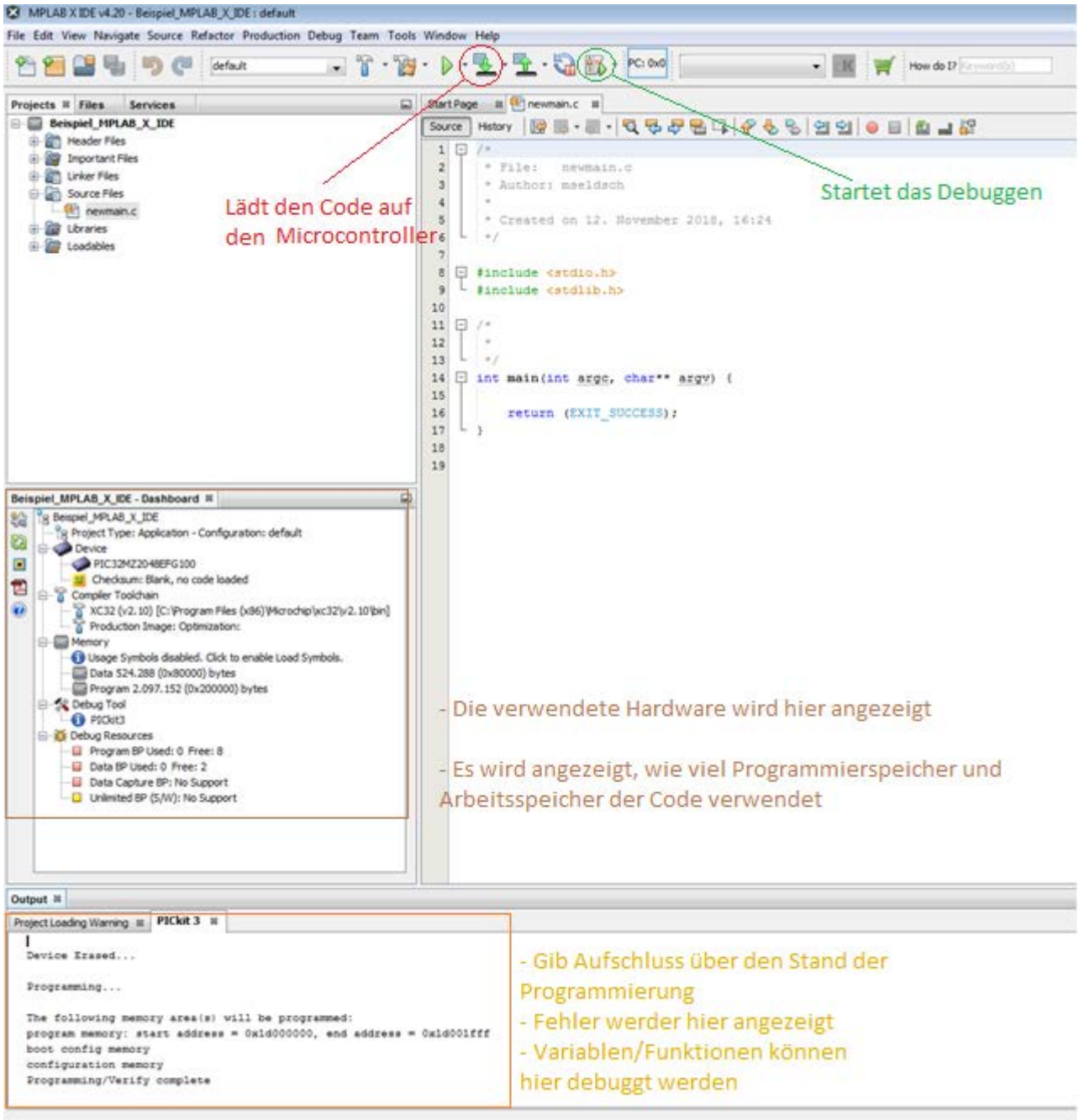

Abbildung 2: MPLAB X IDE- Darstellung

Quelle: Eigene Darstellung – Software: MPLAB X IDE v4.2

MPLAB X IDE bietet eine Vielzahl von Funktionen, dazu gehört das Debuggen, sowie eine Fülle von Plug-Ins, die das Programmieren erleichtern sollen.

3.1.1. Systemvoraussetzung

Die Mindestvorrausetzungen für MPLAB X IDE werden im Folgenden für verschiedene Betriebssysteme aufgeführt.

Microsoft Windows XP Professional SP3/Windows 7 Professional/ Windows 8 Professional:

- Prozessor: Intel Core Duo oder Intel Core 2 Duo
- Arbeitsspeicher: 2 GB (32-bit), 4 GB (64-bit)
- Festplattenspeicher: 1,5 GB

Ubuntu 12.04:

- Prozessor: Intel Core Duo oder Intel Core 2 Duo
- Arbeitsspeicher: 2 GB (32-bit), 4 GB (64-bit)
- Festplattenspeicher: 1,5 GB

Mac OS X 10.8 Intel:

- Prozessor: Intel Core Duo oder Intel Core 2 Duo
- Arbeitsspeicher: 4 GB (32-bit), 4 GB (64-bit)
- Festplattenspeicher: 1,5 GB

Quelle: Microchip - Developer Help [2]

3.2. Programmieradapter

Zur Programmierung des Mikrocontrollers ist es notwendig ein Programmieradapter als Kommunikationsschnittstelle zwischen Computer und Controller zu verwenden. Als Programmieradapter wurde das PICkit 3 ausgewählt. Die Vorteile für diesen Programmieradapter sind die niedrigen Hardwarekosten, der minimale Bedarf an zusätzlicher Hardware für das Debugging und das teure Fassungen oder Adapter nicht benötigt werden, da er sich leicht mit dem beigelegten Mini USB Kabel verbinden lässt.

Abbildung 3: PICKit 3 im Gehäuse

Quelle: Beschriftung selbst erstellt [3]

Die Verbindung mit dem Computer erfolgt über ein Micro USB Kabel. Die Schnittstelle zur Entwicklungsplatine erfolgt über ein sechsadriges Jumper Kabel, welches an der ICSP Schnittstelle, der Entwicklungsplatine vorhanden ist. Je nachdem welche Entwicklungsplatine verwendet wird, kann es auch sein, dass sich ein On Board PICkit3 auf der Platine befindet. Dies ist bei dem ChipKIT-WiFire nicht der Fall, weswegen ein Programmieradapter unabdingbar ist. Bei der Verkabelung ist es wichtig, die Verdrahtung korrekt durchzuführen. Dafür wurde Pin 1 auf dem PICkit3 mit einem Pfeil versehen. Auf dem ChipKIT-Wifire wurde Pin 1 mit einer eckigen Goldumrandung definiert, statt wie gewöhnlich mit einer runden Umrandung. Damit der Kontakt am ChipKIT-WiFire gewährleistet ist, wurden die Einschub Pinne der ICSP Schnittstelle leicht versetzt positioniert, sodass die Pinne unter Druck eingeführt werden und ein plötzliches rausrutschen erschwert wird [3].

3.3. GNSS

GNSS steht für das Globale Navigationssatellitensystem, ein Oberbegriff, der sowohl das GPS der Vereinigten Staaten, das GLONASS von Russland als auch das von Europa entwickelte Ortungssystem Galileo beinhaltet. GNSS wird in diesem Projekt verwendet um eine zeitgenaue Messung zu starten.

3.3.1. GNSS – Puls pro Sekunde

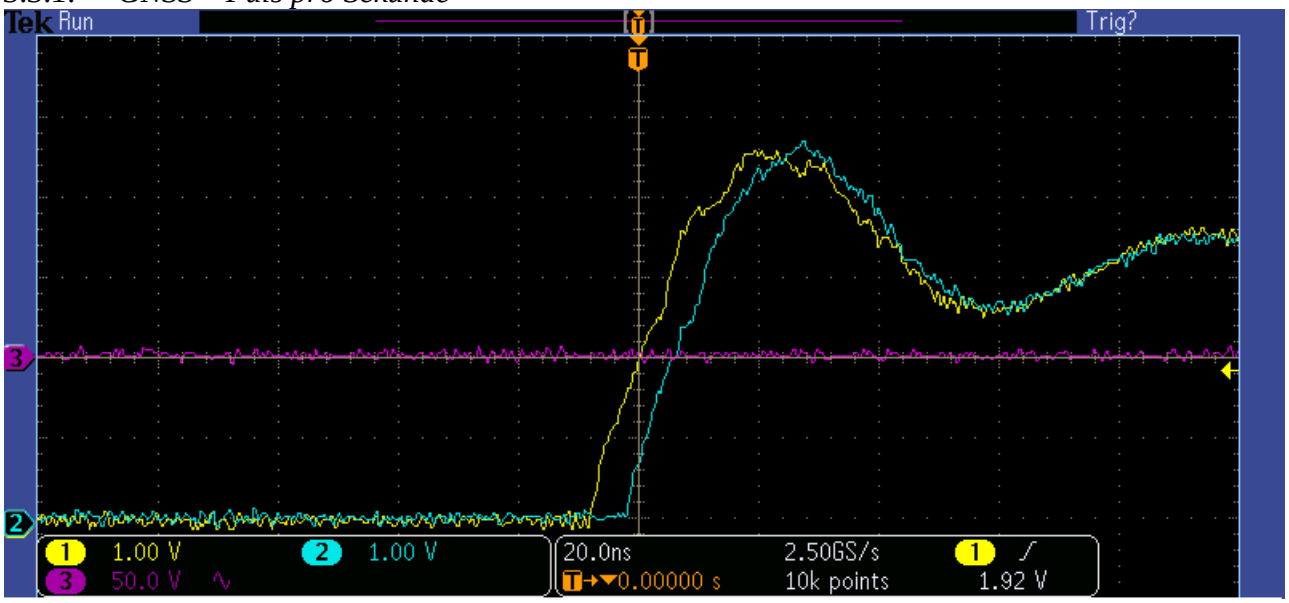

Abbildung 4: GNSS Module – Puls-pro-Sekunde, minimale Differenz untereinander, ca. 8ns

Quelle: Eigene Darstellung

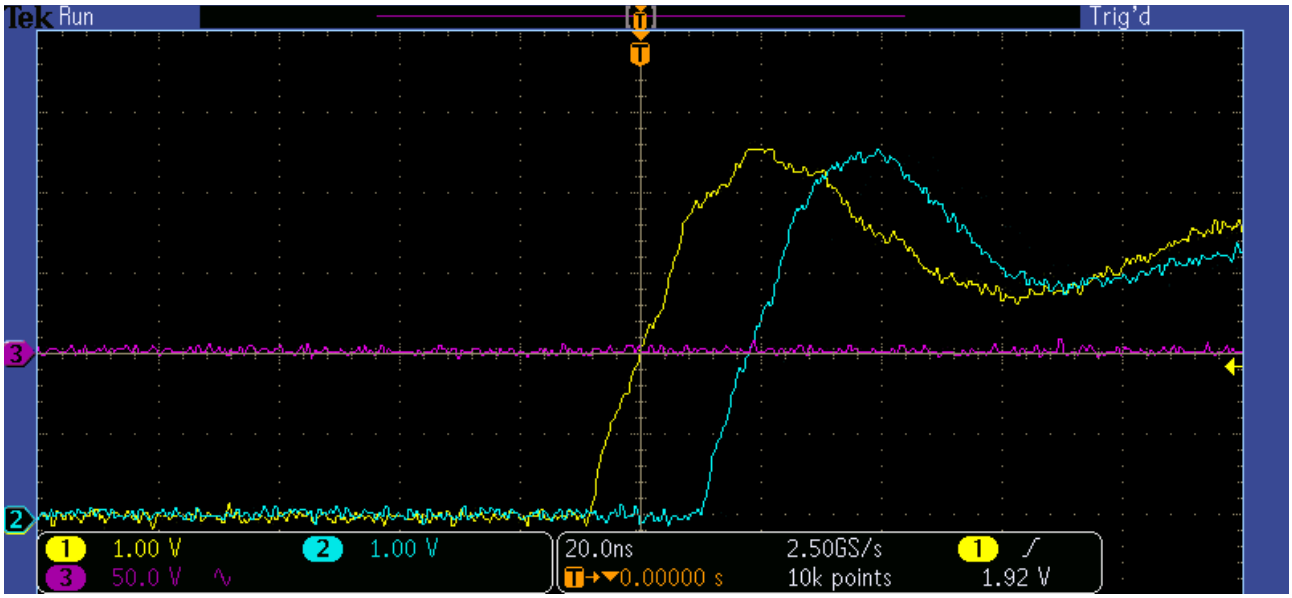

Abbildung 5: GNSS Module – Puls-pro-Sekunde, maximale Differenz untereinander, ca. 20ns

Quelle: Eigene Darstellung

Für das Projekt ist es wichtig, eine Messung zu starten, welche auf ein präzises Zeitsignal erfolgt. Damit beide Messgeräte zur exakt derselben Zeit die Messung starten, startet die Messung, sobald der Puls-pro-Sekunde Takt eingeht. In den Abbildungen 4 und 5 werden die Puls-pro-Sekunde Flanken von zwei GNSS 5 Click Modulen angezeigt. Es wurde verglichen, wie weit die Zeit Abweichungen der GNSS 5 Click Modulen sich unterscheiden. Außerdem wurde ermittelt, wie schnell das Zeitsignal der Puls-pro-Sekunde Flanke eingeht. Wie aus den Abbildung zu erkennen ist, beträgt die Dauer Puls-pro-Sekunde Flanke ca. 30 Nanosekunden. Die Abweichungen der eingehenden Puls-pro-Sekunde Flanken beträgt ca. 5 bis 20 Nanosekunden. Der Puls-pro-Sekunde Takt liefert zu jeder Sekunde ein Signal, von dem NEO-M8N Modul, welches ca. 3,3V beträgt. Über einen analogen Port wird dieses Signal von einen A/D-Wandler gemessen und startet, sobald das Signal eingeht, die Spannungsmessung.

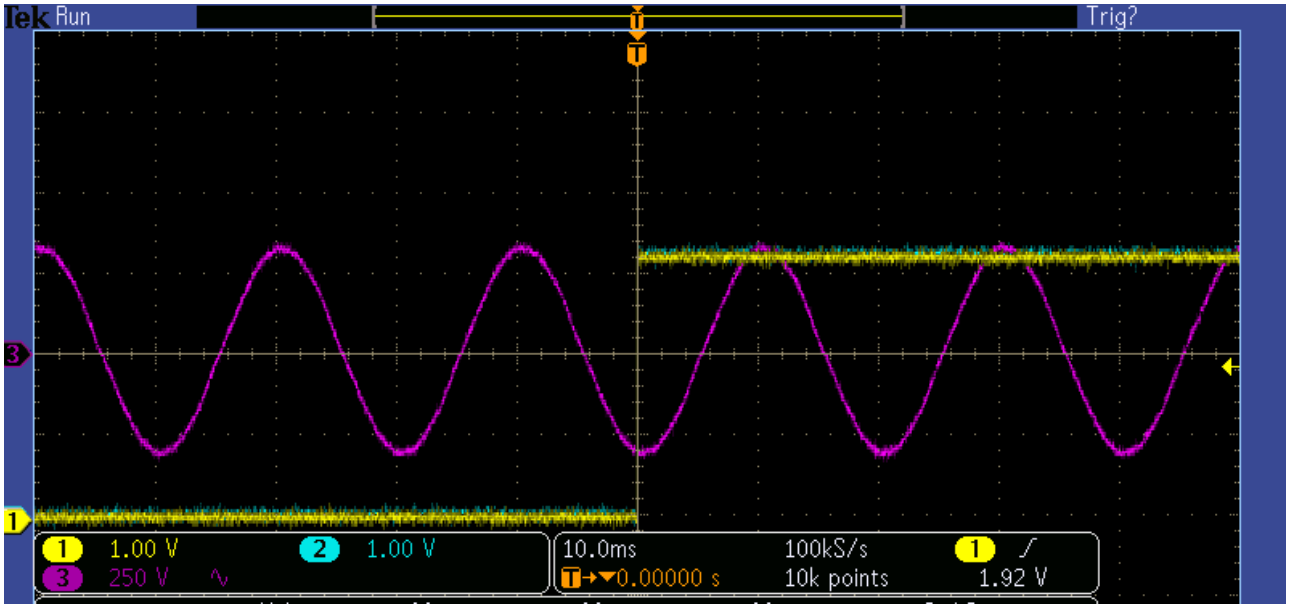

Abbildung 6: GNSS Module – Puls-pro-Sekunde Signal im Verhältnis zur Netzspannung

Quelle: Eigene Darstellung

In Abbildung 6 wird das Signal im Verhältnis zur Netzspannung dargestellt. Man erkennt, dass jenes Puls-pro-Sekunde Signal im Vergleich zur Netzspannung auf der Stelle steht. Das Signal liegt bei einer Länge von ca. 30 Nanosekunden, was einer Abtastgenauigkeit von ca. 1 Millivolt entspricht. Diese Genauigkeit ist mehr als ausreichend, um eine Messung zu starten. Deswegen wurden die GNSS Module als ausreichend für das Projekt empfunden und dementsprechend verwendet.

3.4. A/D-Wandler

Über den A/D-Wandler erfolgt die Spannungsmessung. Der Analog-Digital-Wandler ist eine elektronische Schaltung, die ein analoges Signal in ein digitales umformt. A/D-Wandler wandeln ein zeit-kontinuierliches Eingangssignal in einzelne diskrete Abtastwerte. Diese Abtastwerte werden anschließend in Digitalwerte umgesetzt. Dies passiert mit einer Abtast- Halte- Schaltung (Sample and Hold). Damit nicht zwischenzeitlich neue Werte am AD-Wandler die alten verfälschen wird mithilfe der Sample and Hold Schaltung das Eingangssignal „eingefroren“ und umgewandelt. Anschließend wird der A/D-Wandler für weitere Messungen freigeben. Der A/D-Wandler des PIC32MZ besitzt einen 12-Bit A/D Wandler und übersetzt das analoge Eingangssignal mit der sukzessiven Approximation. Dabei findet eine stufenweise Annäherung statt, mit der man sich Bit für Bit an die zu messende Spannung herantastet [4].

3.5. UART-Schnittstelle

Der Universelle Asynchrone Empfänger und Transmitter (Universal Asynchronus Receiver Transmitter) dient dem Datenaustausch zwischen Computern und Peripheriegeräten. Es ist standardmäßig die serielle Schnittstelle an PCs und Mikrocontrollern. Die UART Schnittstelle wird häufig eingesetzt, um Prozessoren miteinander kommunizieren zu lassen. Das Modul besitzt jeweils eine Schnittstelle zum Datenempfang (RX) und zur Datensendung (TX). Die Datenübertragung findet asynchron und die Rückleiter der UART Module müssen denselben Rückleiter besitzen [5].

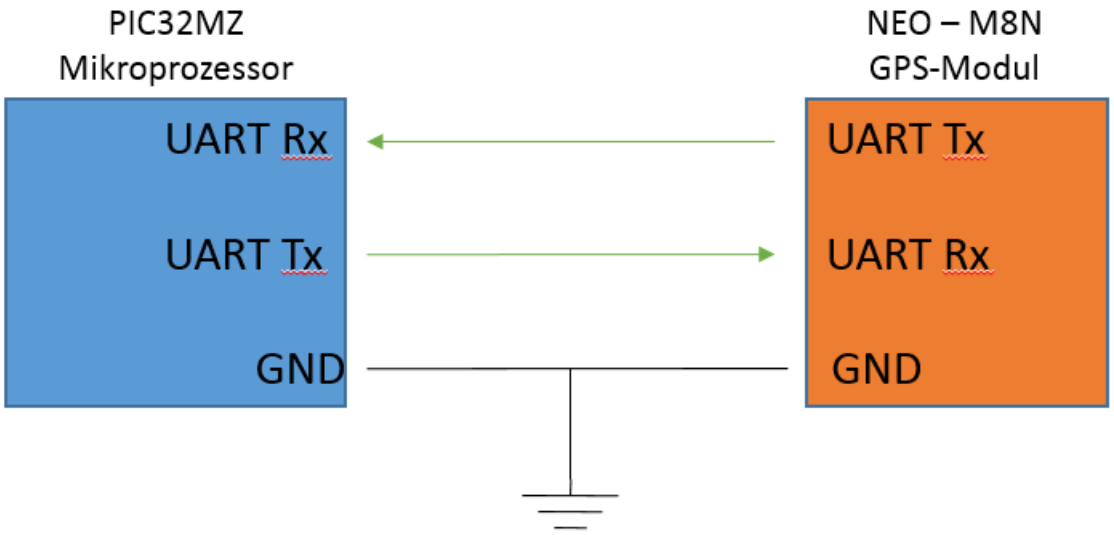

Abbildung 7: Kommunikation zwischen UART Schnittstellen

Quelle: Eigene Darstellung – Angelegt von [4]

3.6. Timer

Mit Hilfe von Timern (Zählern) oder auch Countern ist es möglich in regelmäßigen Zeitabständen Aktionen zu veranlassen. Timer werden in diesem Projekt dazu verwendet die Abtastraten der A/D-Wandler zu konfigurieren. Ein Timer ist ein Register im Mikrocontroller, das hardwaregesteuert fortlaufend um 1 erhöht wird. Dazu wird der Timer mit dem Systemtakt verbunden, um so die Genauigkeit des Quarzes auszunutzen. Sobald das Register gefüllt ist (Overflow) wird dann beispielsweise eine Aktion ausgeführt. Mithilfe eines Vorteilers (Prescaler) kann der Timer vom Systemtakt abweichen und so individuell definiert werden. [7].

3.7. SPI-Schnittstelle

Das Serial Periphere Interface ist ein Bus-System und stellt einen Standard für einen synchronen seriellen Datenbus dar. In diesem Projekt wird die SPI-Schnittstelle zur Kommunikation mit der SD-Karte eingesetzt. Nach dem im Abbildung 8 dargestellten Master-Slave-Prinzip werden diese miteinander verbunden.

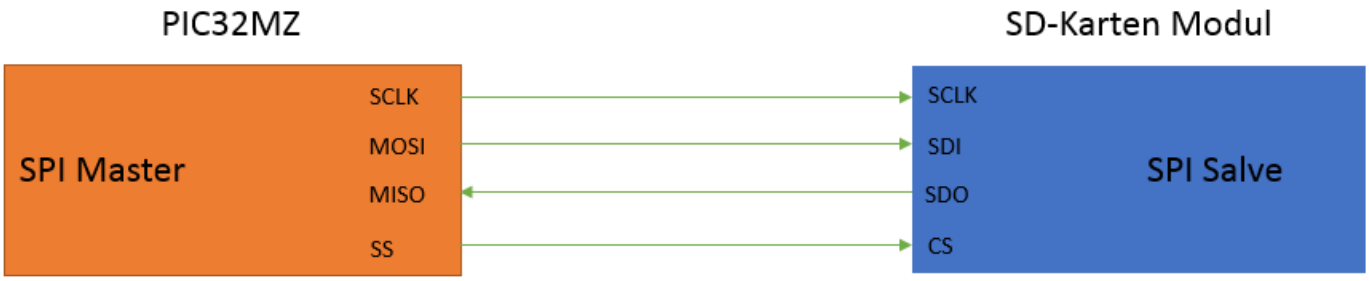

Abbildung 8: Kommunikation zwischen SPI Modulen

Quelle: Eigene Darstellung – Angelegt von [5]

Der Master taktet die Slaves über die SCLK (Serial Clock = Serieller Takt) Schnittstelle und kommuniziert über die MOSI (Master Output, Slave Input = Meister Eingang, Sklave Ausgang) und MISO (Master Input, Slave Output = Meister Ausgang, Sklave Eingang) Schnittstelle. Ein Master kann mit mehreren Slaves kommunizieren, diese werden dann alle über dieselbe SCLK-, MOSI- und MISO-Datenleitung verbunden. Über die SS (Slave Select = Sklave Wählen) Datenleitung wird dann der gewollte Slave angesprochen. Ein SPI Master hat mehrere SS Datenausgänge. Ein Slave hat nur ein CS (Chip Select = Chip Wählen) Dateneingang. Der Slave hat folgende Eingänge: SCLK (Serial Clock = Serieller Takt) und SDI (Serial Data Input = Serieller Daten Eingang): Der Datenausgang heißt SDO (Serial Data Out = Serieller Daten Ausgang) und gibt Daten an den Master weiter [9].

3.8. SD-Karte

Das ChipKIT Wi-FIRE Entwicklungsboard besitzt einen SD-Karten Einschub, welcher genutzt werden kann um Dateien zu speichern oder zu lesen. Die Zeit und die Spannung werden nach abgeschlossener Messung dort gespeichert. Damit der Prozessor darauf zugreifen kann erfolgt die Kommunikation über die SPI3 Schnittstelle [10].

3.9. Interrupts

Ein Interrupt ist eine Funktion, die startet, wenn ein Ereignis eintritt. Der Prozessorzyklus wird vorübergehende unterbrochen, um den Vorgang in der Interrupt-Funktion abzuarbeiten. Wenn die Interrupt-Funktion abgeschlossen ist, setzt der Prozessor seinen Prozess an der unterbrochenen Stelle fort. Die Nutzung von Interrupts ist eine sehr effektive Art, um auf unregelmäßige Ereignisse zu reagieren. Durch den Einsatz von Interrupts wird ein Verlust von Daten verhindert, da die Interrupt-Funktion dann auslöst, wenn an der Schnittstelle Daten empfangen werden. Wenn eine Abfrage im laufenden Programm stattfindet, könnte es passieren, dass der Prozessor sich bei eingehendem Datenempfang in einem anderen Programmabschnitt befindet und somit die dort erreichte Datei verloren gehen kann. Das passiert, weil im Prozessor, sobald er sich wieder an der Schnittstelle befindet, dort bereits eine neue Datei eingegangen seien könnte. Die alte Datei wäre somit verloren. Für das hier programmierte Programm ist es unabdingbar, dass keine Datenverluste entstehen, da es ansonsten zu einer Verfälschung der Messung kommen kann und somit ein Messfehler entsteht, welcher für das Projekt nicht zu vernachlässigen ist [7].

3.10. Register

Ein Mikroprozessor wird über seine Register konfiguriert. Ein Register besteht hier aus bis zu 32 Bits, welche je nach Register unterschiedliche Funktionen aktivieren oder deaktivieren. Jede Hardware des Mikroprozessors beinhaltet in der Regel mehrere Register worüber die Hardware konfiguriert und ausgelesen werden kann. Es gibt zwei Standard Register, welche sich in jeder Komponente befinden.
Über das Konfigurationsregister (abgekürzt CON Register) werden Grundeinstellungen definiert, dass könnte z.B. die Taktfrequenz des Prozessors, die Übertragungsrate der Schnittstellen oder der Auslöser eines Interrupts sein. Die Konfiguration des Konfigurationsregisters wird während des Initialisierungsvorgang eingestellt, da man meist, sobald die Schnittstelle oder das Modul eingeschaltet wird, die Einstellungen nicht mehr verändern kann.
Das Statusregister (abgekürzt STAT Register) wird zumeist in der Endlosschleife des Programms und in Interrupts ausgelesen. Es kann zum Beispiel Aufschluss und Rückmeldung über Fehler in der Hardware oder den Puffer geben. Da ein Mikroprozessor über eine Vielzahl von Komponenten verfügt, welche unterschiedlichste Aufgaben bewerkstelligen können, gibt es noch weitere Register welche speziell für die entsprechenden Komponenten verwendet werden [7].

4. Hardware

4.1. ChipKIT Wi-Fire

Abbildung 9: Entwicklungsplatine – ChipKIT Wi-Fire

Quelle: Datenblatt - ChipKIT Wi-FIRE [6]

Das ChipKIT WiFire verfügt über eine Vielzahl von Anwendungsmöglichkeiten die zusammen mit dem Herzstück, den PIC32MZ2048EFG100 arbeiten. Für eine Reihe von unterschiedlichen Anwendungen stellt die Entwicklungsplatine eine erhebliche Anzahl von Bauteilen zur Verfügung. Die wichtigsten, werden nun erwähnt: Ein Microchip MRF24WG0MA W-LAN Modul, zwei programmierbare Taster, vier programmierbare LED´s, mehrere verschiedene Möglichkeiten zur Spannungsversorgung, eine Micro SD-Karten Schnittstelle, ein Potentiometer, 43 digitale Ein- oder Ausgänge, sowie 12 analoge Eingänge. Durch Positionierung von Jumpern können weiter variable Zustandsänderungen eingestellt werden, z.B. zur Einstellung der Spannungsversorgung oder um eine interne Spannungsquelle auf 3,3V oder 5V einzustellen. Das ChipKIT WiFire kann entweder über die ICSP Schnittstelle, mithilfe eines Programmieradapters unter MPLAB X IDE programmiert werden oder, wenn es mit den dafür entsprechenden Startprogramm (Bootloader) programmiert worden ist, über die serielle Schnittstelle mithilfe der Arduino IDE. Durch diese Vielseitigkeit werden Anwendungsmöglichkeiten für Programmierer geboten, welche auf unterschiedlichste Entwicklungsumgebungen zugreifen, so dass die Entwicklungsplatine für eine Vielzahl von Verbrauchern genutzt werden kann. Im Internet findet man häufig Informationen zu Benutzern, welche die Entwicklungsplatine auf der Arduino Entwicklungsumgebung nutzten. Da in diesem Projekt auf der Entwicklungsumgebung von Microchip programmiert wird, sind diese Beiträge nicht von Belang [10].

4.2. PIC32MZ2048EFG100

Der PIC32MZ2048EFG100 ist ein leistungsstarker 32-bit MCU mit 512KB Arbeitsspeicher, 2MB Flashspeicher und einen 200MHz Takt. Er verfügt über eine Vielzahl von möglichen Schnittstellen, welche, wie im Blockschaltbild zu sehen, über die peripheren Busse angesteuert werden können. Er bietet außerdem eine Vielzahl von Einstellungsmöglichkeiten, dazu gehört z.B. die Variierung des Systemtakts oder die Regulierung der Spannung.

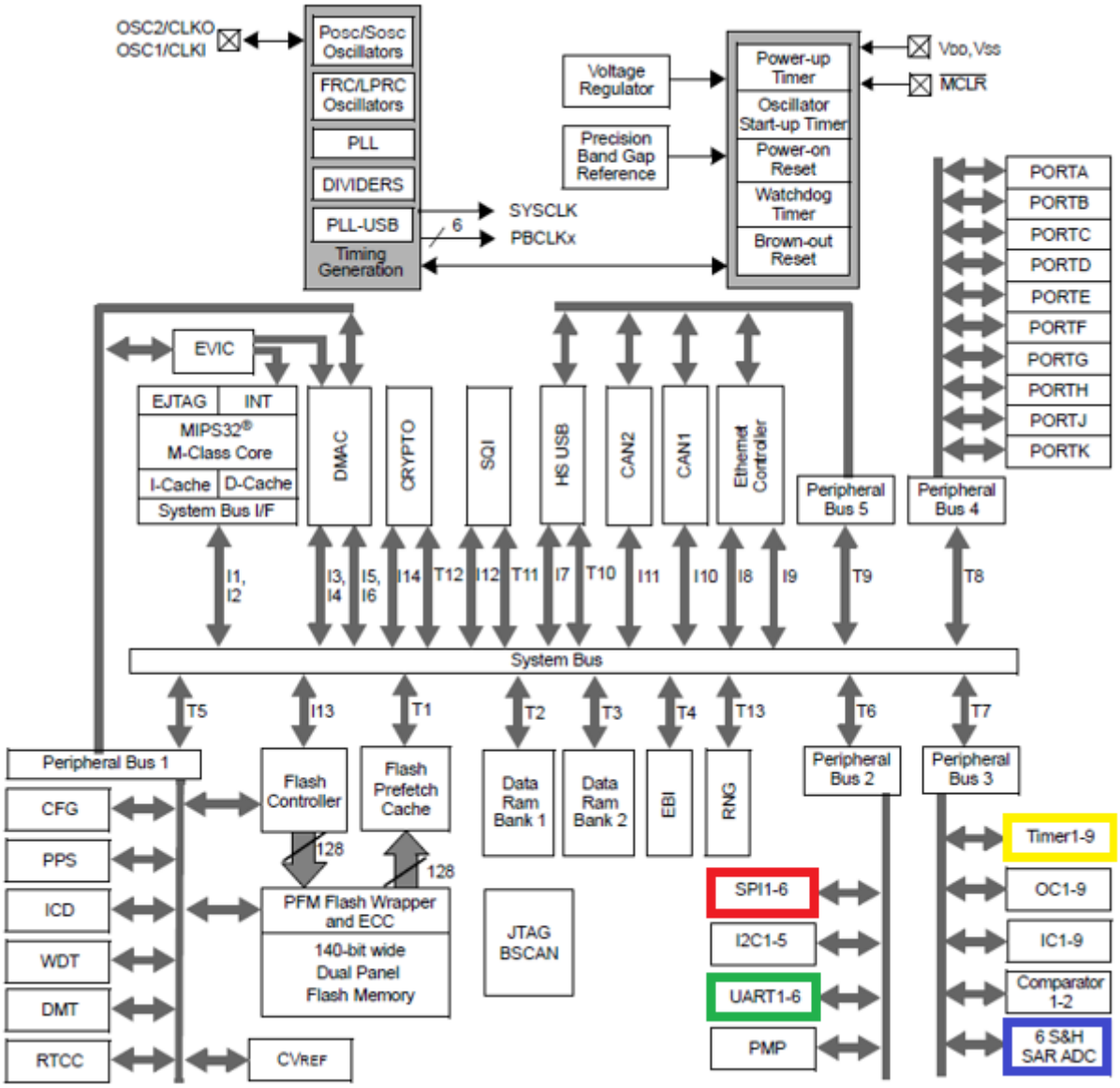

Abbildung 10: Blockschaltbild – PIC32MZ

Quelle: Datenblatt - PIC32MZ [7]

In dem Programm werden die farblich markierten Schnittstellen angesprochen. Die SPI3 Schnittstelle wird für die Verbindung mit SD-Karte benötigt, die UART4 Schnittstelle wird zum Empfang für den GNSS-Zeitempfang verwendet, die AD-Wandler messen das GNSS Puls-pro-Sekunde Taktsignal und die Netzspannung. Timer 3 und Timer 5 werden zur Takteinstellung für den AD-Wandler verwendet [7].

4.3. GNSS 5 Click

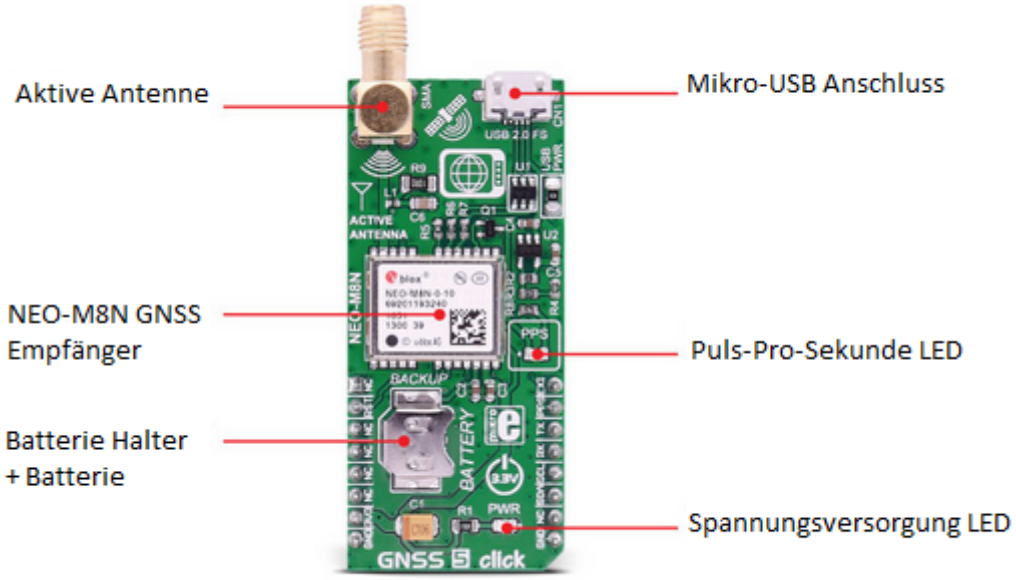

Abbildung 11: GPS Modul - GNSS 5 Click–

Quelle: GNSS 5 Click [8]

Der GNSS 5 Click trägt einen NEO-M8N GNSS Modul, welches den GNSS 5 Click dem Empfang von GNSS Daten ermöglicht. Desweitern verfügt das GNSS 5 Click über benutzerfreundliche LEDs welche dem Benutzer Auskunft über aktuellen Status des GNSS Empfangs und die Spannungsversorgung geben. Die Batterie ermöglicht es, bei einem Ausfall der Versorgungsspannung, alle relevanten Daten im RAM zu speichern. Das GNSS 5 Click Modul kann über einen Mikro-USB-Anschluss auch direkt mit der dazugehörigen Software an einem Computer ausgelesen und mit Spannung versorgt werden [11].

4.4. NEO-M8N

Das NEO-M8N GNSS Modul nutzt den gleichzeitigen Empfang von bis zu drei GNSS-Systemen, erkennt mehrere Konstellationen gleichzeitig und bietet eine hervorragende Positionsgenauigkeit. Dadurch ist eine genauere Ortung an abgeschirmten Ortschaften möglich. Über die SPI-Schnittstelle oder die UART-Schnittstelle können die Daten abgegriffen werden. Über die Schnittstellenauswahl wird definiert, welche Schnittstelle genutzt wird. Standardmäßig befindet sich das NEO-M8N Modul in dem UART Kommunikations-Modus. Das Gerät hält eine maximale Versorgungsspannung von 3,6V aus. Deshalb wird es in der Regel mit dem für Mikroprozessoren üblichen 3,3V betrieben. Das GNSS-Modul sendet jede Sekunde Daten über Datum, Uhrzeit, Tag und Position. Für das Projekt wird eine präzise Uhrzeit benötigt, welche das NEO-M8N Modul mithilfe des GNSS Datenempfangs ermöglicht [11].

5. Software

4.5. Prozessor Konfiguration

4.5.1. Systemtakt und Takteinstellungen

```
/** CONFIGURATION ************************************************************

/* Hiermit wird der Systemtakt auf 200MHz eingestellt */
/* Die System Phasenregelschleife, wird über die folgenden Bits eingerichtet */
#pragma config FPLLICLK = PLL_POSC
// Der Primäre Oszillator wird hierdurch als Eingang gewählt.
#pragma config FPLLIDIV = DIV_3
// Die ankommende Frequenz wir durch 3 geteilt.
#pragma config FPLLRNG = RANGE_5_10_MHZ
// Es wird angegeben in welchen Bereich sich die Frequenz befinden darf.
#pragma config FPLLMULT = MUL_50
// Die Frequenz wird mit 50 multipliziert.
#pragma config FPLLODIV = DIV_2
// Zum Schluss wird die Frequenz noch einmal durch 2 geteilt.


/* Definierung des Externen Oszillators */
#pragma config FNOSC = SPLL
// Die Phasenregelschleife wird über das System variiert.
#pragma config POSCMOD = EC
// Hierdrüber wird der externe Takt als Oszillator eingestellt.
#pragma config UPLLFSEL = FREQ_24MHZ
// Die Externe Takt Frequenz wird auf 24MHz eingestellt.


/* Deadman Timer Konfiguration */
#pragma config FDMTEN = OFF
// Deadman Timer muss deaktiviert werden, da er ansonsten das Programm
stört/unterbricht.


#pragma config ICESEL = ICS_PGx2
// Durch Einstellung dieses Bit wird debuggen ermöglicht
```

Der Systemtakt gibt die Geschwindigkeit des Prozessortakts wieder und wird über die Phasenregelschleife des Systems konfiguriert. Dafür empfängt die System-Phasenregelschleife einen Takt von außerhalb. Ein externer Oszillator, der primäre Oszillator speist den Grundtakt ein. Der Prozessor besitzt mehrere Oszillatoren. Der externe primäre Oszillator, eignet sich wegen seiner Frequenzstabilität für den Systemtakt am besten.

Die System-Phasenregelschleife, wird mit dem FPLL Register eingestellt, sie dient zur Konfiguration des Systemtakts.

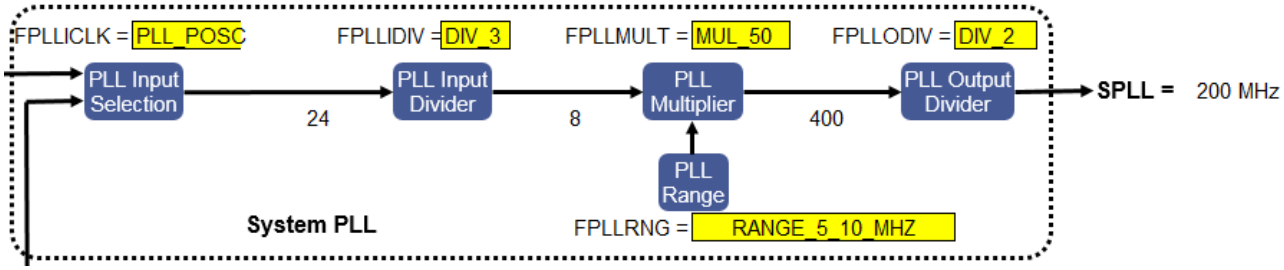

Abbildung 12: PIC32MZ Oszillator Konfiguration - Phasenregelschleife

Quelle: Developer Help - Oszillator Konfiguration [9]

Durch setzen des FPLLICLK Bit wird der externe primäre Oszillator als Taktgeber eingestellt Wie man in Abbildung 12 erkennen kann, können in der Phasenregelschleife verschiedene Frequenzen für das System konfiguriert werden. Für das Phasor-Messgerät wurde die schnellstmögliche Prozessorfrequenz von 200MHz ausgewählt. Diese wurde durch einstellen der folgenden Bits vorgenommen. Der primäre Oszillator gibt einen Takt von 24MHz vor. Der FPLLDIV-Bit teilt die eingehende Frequenz durch drei. Der Takt beträgt nun 8MHz. Das FPLLRNG-Bit konfiguriert den Eingangsbereich für den Multiplizierer der Phasenregelschleife. Das FPLLMULT-Bit wurde so gewählt, dass die eingehende Frequenz mit 50 multipliziert wird. In dem Ausgang der Phasenregelschleife, wird die eingehende Frequenz, welche nun 400MHz beträgt durch das FPLLODIV-Bit mit 2 dividiert. Die Ausgangsfrequenz des Systemtakts wurde dadurch nun auf 200MHz konfiguriert.

Im nächsten Schritt wird der externe primäre Oszillator eingestellt, er dient als Taktgeber der System-Phasenregelschleife.

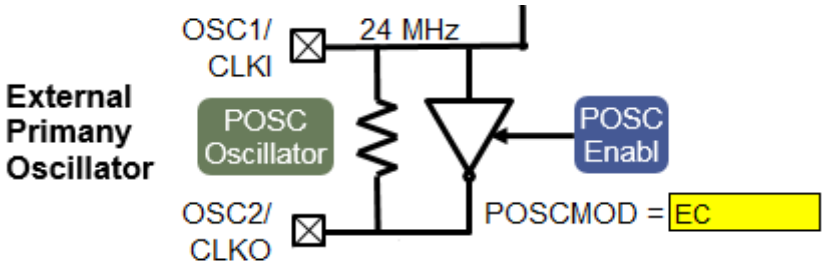

Abbildung 13: PIC32MZ Oszillator Konfiguration – externer primärer Oszillator

Quelle: Developer Help - Oszillator Konfiguration [9]

Mithilfe des FNOSC Bit wird definiert, dass der externe Oszillator die Systemphasenregelschleife ansteuert. Mit dem POSCMODE Bit wird der externe Oszillator aktiviert. Das UPLLFSEL-Bit stellt den Externen Oszillator mit einer Frequenz von 24MHz ein.
Damit das Programm unterbrechungsfrei läuft wird der Deadman-Timer mithilfe des FDMTEN-Bits deaktiviert. Zum Debuggen wurde das ICESEL-Bit mit der entsprechenden Schnittstelle verbunden und eingestellt.

4.5.2. Variablen

```
/** VARIABLEN ****************************************************************/

/* Wird zur Berechnung benötigt, z.B. für die Baud Rate von UART*/
#define SYS_FREQ 200000000 // Running at 200MHz

/* UART/GPS Variablen */
unsigned char A[727];
/* GPS-Daten Variablen */
unsigned char gpstime[6];
/* Zeit Variablen */
unsigned char gpsdate[6];
/* Datum Variablen */
unsigned char gps_header[6]="GNRMC,";
// Der NMEA-Datensatz, welcher die Zeit beinhaltet

/* Festgelegte Zeiten, wann eine Messung erfolgen soll */
unsigned char gps_fest_time1[6]="170210";
unsigned char gps_fest_time2[6]="170220";
unsigned char gps_fest_time3[6]="170230";
unsigned char gps_fest_time4[6]="170240";
unsigned char gps_fest_time5[6]="170250";

/* Hilfsvariablen */
unsigned int a, b, c;
bool string_compare = true;
bool time_catched = true;

/* ADC Variablen */
#define anz 2000
// Definiert die Anzahl, wie viele Messungen pro Scan durchgeführt werden.
int e;
// Zählervariable
int E[anz];
// Speichervariable

/* SD Variablen */
int f,g;
// Hilfsvariablen
unsigned char SaveData[10];
// Namensgebung der gespeicherten Datei
unsigned char CSV[4] = ".csv";
// Definiert den Dateityp, welcher auf die SD Karte gespeichert wird

/* FAT-FS Variablen */
FIL file;
// File handle for the file we open
DIR dir;
// Directory information for the current directory
FATFS fso;
// File System Object for the file system we are reading from
UINT br, bw;
```

Die Variablen werden zur Speicherung von Werten in den vorhandenen Funktionen eingesetzt. Es handelt sich um globale Variablen, welche extra nicht in der Main deklariert wurden, damit sie in jeder Funktion genutzt werden können. In den Kommentaren wird auf die unterschiedliche Verwendung spezieller eingegangen.

4.5.3. Bibliotheken

```
/** INCLUDES ***************************************************************/

#include <sys/attribs.h>
// Enthält Interrupt spezifische Makros, zwingend notwendig um Interrupt Befehle
zu benutzen

#include <xc.h>
// Enthält alle spezifische Prozessorinformationen von Microchip Prozessoren,
erleichtert es ungemein Bits zu verändern und aufzusuchen.

#include <stdio.h>
// Standard-Input-Output-Funktionen

#include <stdbool.h>
// Unterstützung für Boolesche Variablen


#include "ff.h"
// Enthält Funktionen um auf die SD-Karte zuzugreifen.
Ist universell einsetzbar

#include "diskio.h"
// Enthält Prozessorspezifische Einstellungen auf die in ff.h zugegriffen werden
Muss für jeden Prozessor individuell eingestellt/eingerichtet werden.
```

Mithilfe der Bibliotheken wird eine Vielzahl von Funktionen eingebunden, welche das Programmieren erleichtern.

4.5.4. System Konfiguration

```
void SYSTEM_Init(){

SYSKEY = 0xAA996655;
SYSKEY = 0x556699AA;
//SYSKEY Register wird verändert, um Zugriff auf Geräteeinstellungen zu
aktivieren.

/*PB2DIV*/
//Versorgt SPI und UART mit der gewollten Taktfrequenz.
PB2DIVbits.ON = 1;
//Der Periphere Bus 2 Takt wird aktiviert.
PB2DIVbits.PBDIV = 1;
//Hiermit wird die Taktfrequenz von peripheren Bus 2 definiert.

/*PB3DIV*/
//Versorgt Timer und ADC mit der gewollten Taktfrequenz.
PB3DIVbits.ON = 1;
//Der Periphere Bus 3 Takt wird aktiviert.
PB3DIVbits.PBDIV = 1;
//Hiermit wird die Taktfrequenz von peripheren Bus 3 definiert.

/* Die Sequenz wird gesperrt */
SYSKEY = 0x33333333;
//SYSKEY Register wird verändert, um Zugriff auf Geräteeinstellungen zu
deaktivieren.

/* Ordnet dem PIC32MZ die Interrupt Prioritäten zu, dafür wird demensprechend
das Schattenregister definiert */
PRISS = 0x76543210;

/* Setzt den Interrupt Controller in den Multi-Vektor Modus */
INTCONbits.MVEC = 1;

/* Aktiviert Interrupt Ausnahmen */
builtin_enable_interrupts();

/* Initialisiert die LEDs als Output und auf aus. */
LED_LD1 = 0;
TRIS_LED_LD1 = 0;
LED_LD2 = 0;
TRIS_LED_LD2 = 0;
LED_LD3 = 0;
TRIS_LED_LD3 = 0;
LED_LD4 = 0;
TRIS_LED_LD4 = 0;

/* Setzt alle Ports auf Digital */
ANSELA = 0;
ANSELB = 0;
ANSELC = 0;
ANSELD = 0;
ANSELE = 0;
ANSELF = 0;
ANSELG = 0;
}
```

Die Systemkonfigurationen sind für unterschiedliche Einstellungen notwendig. In diesem Projekt werden dort Grundeinstellungen der peripheren Busse und Interrupts konfiguriert.
Damit diese Einstellungen verändert werden können müssen die Register, welche die Einstellungen blockieren, entsperrt werden. Dies wird ermöglicht indem das SYSKEY-Register auf eine bestimmte Zahlenreihe gesetzt wird. Die erste Zahl und die zweite Zahl sind das logische Gegenteil von einander. Es ist eine Sicherheitsfunktion. Nun können die gewünschten Systemeinstellungen verändert werden. Als erstes werden die Taktfrequenzen der verwendeten peripheren Busse auf die schnellstmögliche Taktfrequenz konfiguriert. Die einzelnen Schnittstellen können und werden in den jeweiligen Initialisierungen noch mit einem Vorteiler (Prescaler) konfiguriert. Die peripheren Busse werden auf ihre maximale Frequenz eingestellt. Die peripheren Busse können maximal mit der Hälfte der Prozessorfrequenz eingestellt werden. Bei einer aktuellen Prozessorfrequenz von 200MHz, bedeutet das, dass die peripheren Busse mit 100MHz versorgt werden. Nach diesen Einstellungen werden die Systemeinstellungen wieder gesperrt. Ein weiterer Zugriff auf dies Konfigurationen ist nun nicht mehr möglich. Es finden noch allgemeine Interrupt-Einstellungen statt. Durch Definierung des PRISS-Registers werden die Interrupt-Prioritäten gesetzt, welche von 0 bis 7 gehen. Das MVEC-Bit aus dem Interrupt Konfigurationsregister wird aktiviert und ermöglicht, dass mehrere Interrupts gleichzeitig funktionieren. Die Funktion „builtin_enable_Interrupts“ ist eine prozessorspezifische Anweisung und aktiviert den Zugriff durch Interrupts. Zur Visualisierung wurden im nächsten Schritt alle programmierbaren LEDs als Ausgang deklariert und ausgeschaltet. Damit es nicht zu Konflikten zwischen analogen und digitalen Ports kommt, wurden sicherheitshalber alle auf digital gesetzt. Die verwendeten analogen Ports werden in der AD-Wandler-Konfiguration initialisiert.

4.5.5. Interrupt Konfiguration

```
void INTERRUPT_Init(){

/* Initialisiert ADC0 Daten Interrupt Einstellungen */
// Setzt die Interrupt Priorität auf 5
IPC14bits.ADCD0IP = 5;
// Setzt die Interrupt Subpriorität auf 1
IPC14bits.ADCD0IS = 1;
// Setzt den Interrupt Kennzeichnung auf 0
IFS1bits.ADCD0IF = 0;
// Deaktiviert Interrupts von ADC0
IEC1bits.ADCD0IE = 0;

/* Initialisiert ADC1 Daten Interrupt Einstellungen */
// Setzt die Interrupt Priorität auf 6
IPC15bits.ADCD1IP = 6;
// Setzt die Interrupt Subpriorität auf 1
IPC15bits.ADCD1IS = 1;
// Setzt den Interrupt Kennzeichnung auf 0
IFS1bits.ADCD1IF = 0;
// Deaktiviert Interrupts von ADC1
IEC1bits.ADCD1IE = 0;
```

```
/* Initialisiert UART4 Daten Empfang Interrupt Einstellungen */
// Setzt die Interrupt Priorität auf 4
IPC42bits.U4RXIP = 4;
// Setzt die Interrupt Subpriorität auf 1
IPC42bits.U4RXIS = 1;
// Setzt den Interrupt Kennzeichnung auf 0
IFS5bits.U4RXIF = 0;
// Aktiviert Interrupts von UART4
IEC5bits.U4RXIE = 1;
}
```

Die Interrupt Service Routinen werden benötigt, damit ein zuverlässigerer Datenabgriff gewährleistet wird. Für die spätere Datenanalyse ist es wichtig, dass bei eingehender Messaufnahme, diese konsequent durchgeführt wird. Deshalb hat es sich als sinnvoll ergeben, das Hauptprogramm solange zu unterbrechen, bis die Messungen beendet sind. Über die Interrupt-Service-Routinen werden folgende Messgrößen in der hier dargestellten Reihenfolge abgegriffen:

1. Einkommende GPS-Daten über die UART Schnittstelle (UART4).
2. Der eingehende Puls-Pro-Sekunde Takt des GPS-Moduls über den AD-Wandler (ADC0).
3. Die Spannungsmessung über den AD-Wandler (ADC1).

Bei der Konfiguration der Interrupt-Service-Routinen ist es wichtig, dass die Interrupt-Prioritäten klar gegliedert sind. Da es ansonsten zu Konflikten zwischen den Interrupts kommen kann. Bei allen Interrupts ist es möglich, Subprioritäten zu definieren und somit ein noch variableres Programm zu erstellen. In diesem Projekt hat die Spannungsmessung die höchste Priorität, danach das Takt-Signal der GPS-Flanke und zum Schluss die einkommenden GPS-Daten. Sobald die Interrupt-Priorität geklärt ist, muss die Interrupt-Kennzeichnung auf 0 gesetzt werden, da sich ansonsten das Interrupt bei Aktivierung, sofort in die Interrupt-Service-Routine begibt. Die Interrupt-Kennzeichnung wird immer dann ausgelöst, wenn das Ereignis stattfindet. Anschließend wird das Interrupt aktiviert oder deaktiviert. Im Programm wurde festgelegt, dass die Interrupts zum Empfang der GPS-Flanke und der Spannungsmessung erst bei erfolgreicher Zeitmessung eingeschaltet werden. Somit wird hier vorerst nur das UART-Interrupt aktiviert.

4.6. Schnittstellenkonfiguration

4.6.1. Serielle Periphere Schnittstelle (SPI)

```
void SD_Init(){
/* Auswahl der Peripheren-Pinne, welche mit dem SD-Karten kommunizieren */
SDI3R = 0b0110;
// SDI3 kommuniziert mit RB10.
RPC4R = 0b0111;
// SDO3 kommuniziert mit RC4.
TRISBbits.TRISB10 = 1;
// Setzt den Bit von RB10 auf 1, RB10 wird somit als Input deklariert.
CNPUBbits.CNPUB10 = 1;
//Der Vorwiderstand von RB10 ist ein Pullup-Widerstand und muss entsprechend
deklariert werden.

/* Initialisiert das SPI-Modul und den PIC32MZ für die Kommunikation mit der SD-
Karte. /*
while(disk_initialize(0));

// Bindet die SD-Karte ein. */
f_mount(&fso, "", 0);}
```

Die Funktion SD_Init initialisiert die nötigen Bits, welche für die Kommunikation mit der SD-Karte benötigt werden. Am Anfang werden die entsprechenden Schnittstellen den verwendeten Ports zugewiesen. Auf den PIC32MZ ist es möglich die Ports mit unterschiedlichen Modulen zu verbinden. In der hier dargestellten Funktion werden die Bits, welche mit dem SD-Karten Modul kommunizieren entsprechend verknüpft. Durch das TRIS Register wird definiert, welcher Port, des SD-Karten Moduls als Eingang genutzt wird. Hier ist RB10 mit SDI (Slave Data Input) verbunden. Da vor dem SD-Karten Modul sich ein Pullup-Widerstand befindet, welcher die Betriebsspannug hinaufzieht, muss das mithilfe des CNPU Registers gekennzeichnet werden. Ansonsten kann keine Kommunikation mit dem SD-Karten Modul vorgenommen werden.

```
/* Definiert die Ports entsprechend der Schnittstelle. */
#define CS_SETOUT() TRISCbits.TRISC3 = 0
#define CS_LOW()  LATCbits.LATC3 = 0
#define CS_HIGH() LATCbits.LATC3 = 1

/* Definiert das SPI Register, muss verändert werden, wenn SPI3 nicht verwendet
wird. */
#define SPIBRG SPI3BRG
#define SPIBUF SPI3BUF
#define SPISTATbits SPI3STATbits
#define SPICONbits SPI3CONbits

/* Definiert die Geschwindigkeit der Datenübertragung, hängt von der SD-Karte ab
*/
#define FCLK_SLOW() SPIBRG = 128
//Bei langsamen Takt befindet sich die Frequenz zwischen 100kHz-400kHz.
#define FCLK_FAST() SPIBRG = 16
// Bei schnellem Takt hängt die Geschwindigkeit von der SD-Karte ab.
```

```
void power_on (void){

/* SPI-Einstellungen */
CS_SETOUT();
// Setzt die SPI Schnittstelle als Ausgang fest.
SPICONbits.ON = 0;
// Schaltet das SPI-Modul aus
SPISTATbits.SPIROV = 0;
// Empfangsüberlauf Kennzeichnungsbit
SPICONbits.MSTEN = 1;
// Das SPI-Modul befindet sich in Master Modus
SPICONbits.CKP = 1;
// Takt Polaritätsbit
SPICONbits.CKE = 0;
// SPI Taktflanke Auswahlbit
SPICONbits.SMP = 1;
// Die Eingangsdaten werden am Ende der Datenausgabe eingelesen.
SPICONbits.MODE16 = 0;
SPICONbits.MODE32 = 0;
// Stellt ein über welche Datenübertragungsrate kommuniziert wird.
SPICONbits.SRXISEL = 1;
// Definiert, bei welche Ereignis ein Interrupt ausgelöst werden soll.
SPICONbits.ENHBUF = 0;
// Deaktiviert den erweiterten Puffer
SPICONbits.ON = 1;
//Schaltet das SPI-Modul an.
}
```

Die Datei mmcpic32.c dient der Kommunikation mit einer Multimediakarte. Es ist nötig in den Konfigurationseinstellungen, der mmcpic32.c-Datei, die entsprechende SPI-Schnittstelle zu definieren. Dies geschieht mithilfe der define Instruktion und muss beim Ansprechen einer anderen SPI-Schnittstelle verändert werden.
Die Funktion power on befindet sich in der mmcpic32.c-Datei, welche zur Kommunikation mit der SD-Karte benötigt wird, initialisiert die Serielle-Periphere-Schnittstelle (SPI) und schaltet sie an. Mit CS_SETOUT wird festgelegt, über welchen Pin der Slave angesprochen wird. In dem SPIxCON Register werden die grundlegenden Einstellungen des SPI-Moduls konfiguriert. Damit das SPI-Modul konfiguriert werden kann, wird es ausgeschaltet. Falls sich noch Daten im Puffer befinden, wird mit dem SPIROV-Bit der Puffer geleert. Es wird eingestellt, dass das SPI-Modul des PIC32 als Master fungiert und gibt mithilfe des CKP- und CKE-Bits die Kommunikation vor. Durch Festlegen der Bits MODE16 und MODE23 wird konfiguriert, dass die Datenübertragungsrate 16 Bit beträgt. Das SRXISEL-Bit stellt ein, bei welchem Ereignis ein Interrupt ausgelöst werden soll. Es wird ein Interrupt ausgelöst, sobald der Puffer nicht leer ist. Der erweiterte Puffer Modus wird deaktiviert, da er nicht für die SD-Karten Kommunikation benötigt wird. Zum Schluss wird das SPI-Modul aktiviert.

4.6.2. Analog-Digital-Umsetzer (A/D)

```
void InitializeADC(){

ANSELBbits.ANSB5 = 1;
// Port B5 wird als analog deklariert, ADC0 empfängt hierüber seine Daten.
ANSELBbits.ANSB9 = 1;
// Port B9 wird als analog deklariert, ADC1 empfängt hierüber seine Daten.

/* Initialisiert die ADC-Kalibrierungswerte anhand der werkseitigen
programmierten DEVADCx Flash Register */
ADC0CFG = DEVADC0;
ADC1CFG = DEVADC1;

/* Konfiguration des ADC Kontrollregister 1 */
ADCCON1 = 0;
// Alle Features werden deaktiviert.

/* Konfiguration des ADC Kontrollregister 2 */
ADCCON2 = 0;
// Alle Features werden deaktiviert.

/* Konfiguration des Aufwärme Registers */
ADCANCON = 0;
// Alle Features werden deaktiviert.
ADCANCONbits.WKUPCLKCNT = 5;
// Definiert wie viele Takt Zyklen ablaufen sollen, bis der AD-Wandler
einsatzbereit ist.

/* Konfiguration des ADC Kontrollregister 3 */
ADCCON3 = 0;
// Alle Features werden deaktiviert.
ADCCON3bits.ADCSEL = 0b00;
// Als Taktgeber wird PBCLK3 ausgewählt.
ADCCON3bits.CONCLKDIV = 0b000001;
// Definiert, dass der Steuerungstakt die hälft des Systemtakts entspricht.
ADCCON3bits.VREFSEL = 0;
// Die positive Spannung (VDD) und die negative Spannung (VSS) wird als
Referenzspannung ausgewählt.

/* Konfiguration der ADC Abtastzeit und des Taktgebers */
ADC0TIMEbits.ADCDIV = 1;
// ADC0 Takt Frequenz entspricht der Hälfte des Steuerungstakts.
ADC0TIMEbits.SAMC = 5;
// Die Abtastzeit beträgt das 5-fache der des Steuerungstakts.
ADC0TIMEbits.SELRES = 3;
// Die Auflösung von ADC0 beträgt 12 Bits.
ADC1TIMEbits.ADCDIV = 1;
// ADC1 Takt Frequenz entspricht der Hälfte des Steuerungstakts.
ADC1TIMEbits.SAMC = 5;
// Die Abtastzeit beträgt das 5-fache der des Steuerungstakts.
ADC1TIMEbits.SELRES = 3;
// Die Auflösung von ADC0 beträgt 12 Bits.

/* Definiert den Analogen Port, worauf ADC0 und ADC1 zugreifen */
ADCTRGMODEbits.SH0ALT = 0b00;
// ADC0 = AN0
ADCTRGMODEbits.SH1ALT = 0b00;
// ADC1 = AN1
```

```
/* Konfiguration des Steuerregisters, welches den Eingangsmodus definiert */
ADCIMCON1bits.SIGN0 = 0;
// Es wird ein Datenformat ohne Vorzeichen gewählt.
ADCIMCON1bits.DIFF0 = 0;
// Der betrieb findet im Single-Ended Betrieb statt
ADCIMCON1bits.SIGN1 = 0;
// Es wird ein Datenformat ohne Vorzeichen gewählt.
ADCIMCON1bits.DIFF1 = 0;
// Der Betrieb findet im Single-Ended Betrieb statt

/* Konfiguration des Interrupt Registers */
ADCGIRQEN1bits.AGIEN0 = 1;
// Das Interrupt für ADC0 wird generiert, sobald die konvertierte Datei bereit
ist.
ADCGIRQEN1bits.AGIEN1 = 1;
// Das Interrupt für ADC1 wird generiert, sobald die konvertierte Datei bereit
ist.

/* Konfiguration des Trigger Registers */
ADCTRG1bits.TRGSRC0 = 0b00110;
// AN0 wird auf das Signal von Timer3 getriggert.
ADCTRG1bits.TRGSRC1 = 0b00111;
// AN1 wird auf das Signal von Timer5 getriggert.

/* Schaltet die ADC Module an */
ADCCON1bits.ON = 1;

/* Wartet bis die Referenzspannung an den ADC Modulen stabil ist */
while(!ADCCON2bits.BGVRRDY);
while(ADCCON2bits.REFFLT);

/* Aktiviert den Takt für den Analogen Kreislauf von ADC0 und ADC1 */
ADCANCONbits.ANEN0 = 1;
ADCANCONbits.ANEN1 = 1;

/* Wartet bis ADC0 und ADC1 bereit sind */
while(!ADCANCONbits.WKRDY0);
while(!ADCANCONbits.WKRDY1);
/* Aktiviert die ADC Module */
ADCCON3bits.DIGEN0 = 1;
// Aktiviert ADC0
ADCCON3bits.DIGEN1 = 1;
 // Aktiviert ADC1

}
```

Der Analoge Digitale Umsetzer, wandelt ein analoges Signal in ein digitales um. Da der A/D-Wandler vielseitig einsetzbar ist, ist es notwendig die Konfigurationsbits entsprechend der gewünschten Auslegung einzustellen. Die A/D-Wandler werden verwendet, um die Spannungsmessungen auszuführen. Dafür werden die A/D-Wandler 0 und 1 verwendete (ADC0 & ADC1). ADC0 misst die den Puls der GNSS-Taktflanke. ADC1 kümmert sich um die Spannungsmessung, welches im weiteren Verlauf gespeichert und ausgewertet wird. Es ist wichtig, dass die entsprechenden Ports als analoge deklariert werden, da sie der Prozessor sonst als digitale Eingänge deklariert. In digitalen Eingängen/Ausgängen können nur zwei verschiedene Werte abgegriffen werden (0 und 1). An analogen Eingängen/Ausgängen können, je nachdem mit viel Bit der A/D-Wandler arbeitet, mehrere verschieden Werte abgegriffen werden. Es steht uns hier ein A/D-Wandler mit 12-Bit zur Verfügung. Damit können 4096 (2^12) unterschiedliche Werte dargestellt werden. Mithilfe des ANSEL Registers wird konfiguriert, ob es sich um einen analogen (1) oder digitalen (0) Pin handelt. Das ADC-Kontrollregister 1 und 2 wird für eine Spannungsmessung im mittelschnellen Bereich nicht benötigt und wird deshalb ausgeschaltet. Anschließend wird das Konfigurationsregister, welches die Aufwärmzeit des ADC-Moduls definiert, eingestellt. Es hat sich gezeigt, dass 32 Systemtakte ausreichend sind. Dies entspricht der dezimalen Darstellung von 5. Als nächstes wird das ADC-Kontrollregister 3 konfiguriert. Alle Register werden von Anfang an ausgeschaltet und danach die gewünschten Funktionen aktiviert. Als erstes wird der periphere Bus 3 als Taktgeber konfiguriert. Im nächsten Bit wird definiert, dass der Steuerungstakt der Hälfte des Systemtakts entsprechen soll. Zuletzt wird als Referenzspannung die Versorgungsspannung des Mikrocontrollers ausgewählt. Das nächste Register konfiguriert die Abtastzeit und den Taktgeber. Beide ADC-Module wurden mit der maximalen Taktfrequenz konfiguriert. Dies entspricht der halben Taktfrequenz des Steuerungstakts. Die Abtastzeit wurde auf das 5-fache des Steuerungstakts festgelegt, was einer Abtastfrequenz von 20MHz entspricht. Anschließend wurde die Genauigkeit der AD-Wandler mit einer maximalen Auflösung von 12 Bit konfiguriert.
Im nächsten Abschnitt wird der Eingangsmodus festgelegt. Das Spannungssignal wird im Single-Ended Betrieb abgegriffen. Das heißt, dass der Eingang mit Bezug zur Systemmasse betrieben wird und die Störsignale gemeinsam mit dem Signal erfasst werden. Als empfangener Datentyp wird ein Datentyp ohne Vorzeichen gewählt. Damit die umgewandelten analogen Daten gespeichert werden können, wird im nächsten Register die Interrupt Einstellungen konfiguriert, welche festlegen, auf welches Ereignis ein Interrupt ausgelöst werden soll. Es wurde ausgewählt, dass jenes Interrupt bei fertiger Konvertierung ausgelöst werden soll. Danach werden die ADC-Module auf den Takt von Timer 3 und Timer 5 konfiguriert, welche die Abtastrate definieren. Nachdem die Konfiguration feststeht werden die ADC-Module eingeschaltet. Es wird nun abgewartet, bis sich die Referenzspannung stabilisiert hat. Anschließend werden die entsprechende ADC-Modul mit dem vordefinierten Takt versorgt. Sobald ADC0 und ADC1 bereit sind, werden sie dann aktiviert.

4.6.3. Universeller Asynchroner Empfänger Sender (UART)

```
void UART_Init(){

//Die Variable pbClk hängt von der Systemfrequenz ab. In den
Systemkonfigurationen wurde eingestellt, das PBCLK mit höchstmöglicher Frequenz
laufen soll, welches der halben Systemfrequenz entspricht.
int pbClk;
pbClk = SYS_FREQ / 2;

//Alle Register von UART4 werden zur Einstellung auf 0 gestellt.
U4MODE = 0;
//Der Geschwindigkeitsmodus wird auf den Standard Modus eingestellt, da keine
schnelle Datenübertragung benötigt wird.
U4MODEbits.BRGH = 0;
//Die Baud Rate wird anhand der Formel aus dem Datenblatt definiert.
U4BRG = (pbClk / ((16 * 9600) - 1));
//Die Status Register Bits werden auf 0 gesetzt.
U4STA = 0;
//Aktiviert den Bit zum Datenempfang.
U4STAbits.URXEN = 1;
//Ein Interrupt soll, dann auslösen, sobald 1 Daten Charakter sich im Puffer
befindet.
U4STAbits.URXISEL = 0b00
//Das PDSEL Register kontrolliert wie viele Daten Bits und Paritätsbits benötigt
werden.
U4MODEbits.PDSEL = 0;
//Das STSEL Register kontrollier wie Stopp Bits verwendet werden.
U4MODEbits.STSEL = 0;
//Nach den Einstellungen wird das UART4 Modul eingeschaltet
U4MODEbits.ON = 1;
}
```

Die Universelle Asynchrone Schnittstelle dient zum Empfangen und Versenden von Daten. Für den hier dargestellten Anwendungsfall wird nur der Empfang von Daten benötigt. Das NEO-M8N GPS-Modul wird über die UART Schnittstelle ausgelesen oder angesprochen. Dafür müssen spezielle Register entsprechend dem GPS-Modul konfiguriert werden. Wichtig ist es, dass Empfänger und Sender über dieselbe Baud Rate kommunizieren. Das GPS-Modul gibt an seinem UART Ausgang folgende Konfiguration vor:

9600 Baud, 8 Bits, Keine Parität, 1 Stopp Bit.

$$UxBRG = \frac{PBCLK}{16 * Baud\ Rate} - 1$$

$UxBRG$ = UARTx Baud Rate Register, $PBCLK$ = Periphere Bus Clock (Takt),
$Baud\ Rate$ = Gewünschte Baud Rate

In dem MODE Register des UART-Moduls können die benötigten Bits entsprechend konfiguriert werden. Mit dem BRG-Bit wird die Baud Rate mithilfe der oben dargestellten Formel eingestellt. Das PDSEL-Bit wurde mit 0 konfiguriert, dies entspricht einem 8-Bit Datentransfer und keinem Paritätsbit. Durch setzten des STSEL-Registers auf 0 wurde das das Stopp-Bit auf 1 eingestellt.

4.6.4. Timer

```
void TIMER_Init(){

/* Einstellungen für Timer 3, taktet ADC0 */
//Der Timer wird ausgestellt.
T3CONbits.TON = 0;
//Der Timer befindet sich in 16-bit Modus.
T3CONbits.T32 = 0;
//Nun wird der Vorteiler (Prescaler) des Timers definiert.
T3CONbits.TCKPS = 0b000;
//Mithilfe des Perioden Register wird der Timertakt auf 50MHz eingestellt.
PR3 = 1;
//Timer Count Register.
TMR3 = 0;

/* Einstellungen für Timer 5, taktet ADC1*/
//Der Timer wird ausgestellt.
T5CONbits.TON = 0;
//Der Timer befindet sich in 16-bit Modus.
T5CONbits.T32 = 0;
//Der Vorteiler (Prescaler) des Timers wird eingestellt (.
T5CONbits.TCKPS = 0b111;
//Mithilfe des Perioden Register wird der Timertakt auf 16kHz eingestellt.
PR5 = 23;
//Timer Count Register.
TMR5 = 0;


//Schaltet die Timer an.
T3CONbits.TON = 1;
T5CONbits.TON = 1;
}
```

In den Timer-Konfigurationsregistern werden die Timer zur Konfiguration ausgeschaltet und in 16-Bit Modus initialisiert. Das Perioden-Register (PR) kann somit bis maximal 65535 ((2^16) -1) hochgezählt werden.
Die gewünschte Frequenz kann mit folgender Formel eingestellt werden.

$$PRI = \frac{PBCLK}{PS * fg} - 1$$

PRI = Perioden Register, $PBCLK$ = Peripherer Bus Takt (Periphere Bus Clock),
PS = Vorteiler (Prescaler), fg = Frequenz gesucht

Anhand der oben beschriebenen Formel wurden dann die Frequenzen eingestellt. Timer 3 greift mit 50MHz den Puls der eingehenden GPS-Flanke ab. 50 MHz ist die schnellstmögliche Taktfrequenz und wurde dementsprechend eingestellt, weil die eingehende Flanke sich im Nano Sekunden Bereich befindet. Timer 5 wurde mit 16kHz definiert, was einer Abtastfrequenz von 62,5µs entspricht. Bei einer Netzfrequenz von 50Hz und einem Wertebereich von ca. 650V werden somit ca. alle 40mV Messungen aufgenommen. Dies entspricht ca. 16000 Messungen pro Netzperiode. Diese Genauigkeit ermöglicht eine hohe Genauigkeit zur Darstellung des Spannungsverlaufs.

4.7. Funktionen

4.7.1. Interrupt Service Routinen

4.7.1.1. UART4 Interrupt

```
/*UART4 - Signal Empfangen Interrupt */
void __ISR_AT_VECTOR(_UART4_RX_VECTOR, IPL4SRS) UART4RD_Interrupt(){

//Vergleicht, ob der Puffer übergelaufen ist
if(U4STAbits.OERR == 1){
//Löscht den Puffer, sodass weiterer Datenempfang möglich ist.
U4STAbits.OERR = 0;
}
LED_LD2_INV()
//Speichert die empfangenen Daten in einem Array
A[a] = U4RXREG;
a++;
//Setzt die Interrupt Kennzeichnung zurück, weiterer Interrupt Empfang
ermöglicht.
IFS5bits.U4RXIF = 0;

}
```

Die Interrupt Service Routine (ISR), welche die eingehenden GPS Daten empfängt, springt an, sobald sich Daten im Datenpuffer befinden. In der ISR, wird als erstes verglichen, ob der Puffer übergelaufen ist, falls dass der Fall ist wird den Puffer geleert, damit eine weiterer Daten Empfang möglich ist. Anschließend werden die empfangenen Daten in einem Array, welches den Datentyp Charakter besitzt, abgespeichert. Am Ende wird das Interrupt Kennzeichnungsbit (Flag) auf 0 (Low) gesetzt. Dadurch wird die ISR beendet. Sobald sich eine Datei wieder im UART Puffer befindet, wird dieser Bit auf 1 (High) gesetzt und die ISR fängt wieder von vorne an. Bei nicht Setzung dieses Bits beendet sich die ISR nie.

4.7.1.2. ADC0 Interrupt

```
/*ADC DATA0 Interrupt*/
void __ISR_AT_VECTOR(_ADC_DATA0_VECTOR, IPL5SRS) ADC0_Interrupt(){

/* Startet, wenn die GPS-Flanke eingeht*/
if(ADCDATA0>3000){
IEC1bits.ADCD0IE = 0;
//Deaktiviert Interrupts von ADC0
IEC1bits.ADCD1IE = 1;
//Aktiviert Interrupts von ADC1
}
/* Löscht die Interrupt Kennzeichnung */
IFS1bits.ADCD0IF = 0;

}
```

Die ADC0 Interrupt Service Routine (ISR), empfängt die eingehende GPS Flanke, sobald Interrupts für das ADC0 Modul aktiviert werden. Das Interrupt wird generiert sobald die konvertierte Datei bereit ist. Die Datei wird konvertiert, sobald der Trigger ausgelöst wird. Der Trigger ist anhand von Timer 3 auf die gewünschte Abtastfrequenz definiert. In der ISR wird verglichen ob die konvertierte Datei bereit ist, anschließend wird geprüft ob die GPS-Flanke empfangen wird. Falls dies der Fall ist, wird die ISR für ADC0 deaktiviert und die ISR für ADC1 aktiviert. Nun kann die Spannungsmessung beginnen. Zum Schluss wird das ISR Kennzeichnungsbit auf 0 gesetzt.

4.7.1.3. ADC1 Interrupt

```
/*ADC DATA1 Interrupt*/
void __ISR_AT_VECTOR(_ADC_DATA1_VECTOR, IPL5SRS) ADC1_Interrupt(){

/* Speichert das Resultat */
E[e] = ADCDATA1;
e++;

/* Wenn die Messung abgeschlossen ist */
if(e == anz){
IEC1bits.ADCD1IE = 0;
// Deaktiviert Interrupts von ADC1
}
/* Löscht die Interrupt Kennzeichnung */
IFS1bits.ADCD1IF = 0;

}
```

Die ADC1 Interrupt-Service-Routine (ISR) führt die Spannungsmessung durch und wird ausgelöst, sobald eine konvertierte Datei bereit ist. Die Abtastzeit wurde mithilfe von Timer 5 festgelegt. Sobald eine konvertierte Datei bereit steht wird diese in dem Array E, Datentyp Integer, gespeichert. Wenn die Messungen die gewünschte Anzahl erreicht hat, welche durch die Variable anz (Anzahl der Spannungsmessungen) definiert wird, wird das Interrupt für ADC1 deaktiviert. Zu Schluss wird die Interrupt-Kennzeichnung gelöscht, so dass bei weiterem Datenempfang das Interrupt wieder ausgelöst wird, falls es aktiviert wurde.

4.7.2. UART Read

```
void UART_Read(){
    if(a==727){
    // Starte, sobald der GPS-Datensatz voll ist.
    IEC5bits.U4RXIE  = 0;
    // Deaktiviert Interrupts von dem UART-Modul

    /* Geht den kompletten GPS-Datensatz durch auf der Suche nach GNRMC, */
    for(b=0; b<=727; b++){

        if(A[b] == '$'){
            b++;
            if(A[b] == 'G'){
                b++;
                if(A[b] == 'N'){
                    b++;
                    if(A[b] == 'R'){
                        b++;
                        if(A[b] == 'M'){
                            b++;
                            if(A[b] == 'C'){
                                b++;
                                if(A[b] == ','){
                                    b++;
                                    string_compare = true;
                                    break; //Ende, for Schleife
                                }// Ende, if „,“ Bedingung
                            }// Ende, if „C“ Bedingung
                        }// Ende, if „M“ Bedingung
                    }// Ende, if „R“ Bedingung
                }// Ende, if „N“ Bedingung
            }// Ende, if „G“ Bedingung
        }// Ende, if „$“ Bedingug
    }// Ende, for Schleife

        if (string_compare == true){
/* Die Empfangene Zeit wird gespeichert */
                for (c=0;c<6;c++) {
                    gpstime[c] = A[b];
                    b++;
                }
/* Das Empfangene Datum wird gespeichert */
                b = b+40;
                for (c=0;c<6;c++){
                    gpsdate[c] = A[b];
                    b++;
                }
```

```
/* Vergleicht die aktuelle Zeit mit der festgelegten Zeit */
                for (c=0;c<6;c++){
                    if(gpstime[c] == gps_fest_time1[c]){
                        string_compare = true;
                    }
                    else{
                        string_compare = false;
                        break;
                    }
                }
/*Wenn die aktuelle Zeit der festgelegten Zeit entspricht, startet Time
Reached*/
                    if(string_compare==true){
                        Time_Reached();
                    }
/* Vergleicht die aktuelle Zeit mit der festgelegten Zeit */
                for (c=0;c<6;c++){
                    if(gpstime[c] == gps_fest_time2[c]){
                        string_compare = true;
                    }
                    else{
                        string_compare = false;
                        break;
                    }
                }
/*Wenn die aktuelle Zeit der festgelegten Zeit entspricht, startet Time
Reached*/
                    if(string_compare==true){
                        Time_Reached();
                    }

/* Vergleicht die aktuelle Zeit mit der festgelegten Zeit */
                for (c=0;c<6;c++){
                    if(gpstime[c] == gps_fest_time3[c]){
                        string_compare = true;
                    }
                    else{
                        string_compare = false;
                        break;
                    }
                }
/*Wenn die aktuelle Zeit der festgelegten Zeit entspricht, startet Time
Reached*/
                    if(string_compare==true){
                        Time_Reached();
                        }
/* Vergleicht die aktuelle Zeit mit der festgelegten Zeit */
                for (c=0;c<6;c++){
                    if(gpstime[c] == gps_fest_time4[c]){
                        string_compare = true;
                    }
                    else{
                        string_compare = false;
                        break;
                    }
                }
```

```
/*Wenn die aktuelle Zeit der festgelegten Zeit entspricht, startet Time
Reached*/
                    if(string_compare==true){
                        Time_Reached();
                        }
/* Vergleicht die aktuelle Zeit mit der festgelegten Zeit */
                for (c=0;c<6;c++){
                    if(gpstime[c] == gps_fest_time5[c]){
                        string_compare = true;
                    }
                    else{
                        string_compare = false;
                        break;
                    }
                }
/*Wenn die aktuelle Zeit der festgelegten Zeit entspricht, startet Time
Reached*/
                    if(string_compare==true){
                        Time_Reached();
                        }
        }
/* Startet UART */
    UART_Start();
    }
}
```

Die UART Read Funktion sichtet die benötigten Daten aus dem Empfang des GPS-Datensatzes. Dabei wird zuallererst abgefragt, ob der GPS-Datensatz komplett ist. Während des Programmierens hat sich ergeben, dass ein GPS Datensatz aus 727 Zeichen besteht. Sobald dieser erreicht ist, wird das Interrupt, welches das UART Modul ausliest, deaktiviert und anschließend wird nach der Zeichenkette „GNRMC“ gesucht. Die Zeichenkette „GNRMC“ enthält die aktuelle Zeit sekundengenau und das aktuelle Datum. Wenn die Zeit empfangen worden ist, wird die for-Schleife mit der booleschen Variablen string_compare beendet, so dass das Auslesen des GPS-Datensatzes beendet wird da keine weiteren Informationen benötigt werden. Im nächsten Schritt wird nun verglichen, ob die aktuelle Zeit einer der festgelegten Zeiten entspricht. Wenn dies der Fall ist, wird die variable string_compare auf wahr(true) gesetzt. Zurzeit werden 5 festgelegte Zeiten abgefragt (erweiterbar). Falls die aktuelle Zeit der festgelegten Zeit entspricht, wird die Funktion Time_Reached gestartet. Falls die aktuelle Zeit nicht der festgelegten Zeit entspricht, wird die variable string_compare auf falsch(false) gesetzt. Die Interrupts für das UART-Modul werden aktiviert und die Funktion beendet sich.

4.7.3. Time Reached

```
void Time_Reached(){
    IEC1bits.ADCD0IE = 1;
// Aktiviert Interrupts von ADC0, leitet die Spannungsmessung ein
    time_catched = false;
// Setzt die Bedingung auf falsch, sodass UART Interrupts nicht dazwischenfunken
}
```

Die Funktion Time_Reached Funktion befindet sich in der UART_Read-Funktion und wird ausgeführt, wenn die aktuelle Zeit der festgelegten Zeit entspricht. Sie aktiviert Interrupts von ADC0. Die boolesche Variable time_catched wird auf 0 (false) gesetzt. Sie sorgt dafür, dass Interrupts von UART nicht mehr aktiviert werden. Dadurch werden Konflikte zwischen den Interrupts vermieden.

4.7.4. UART Start

```
void UART_Start(){
    a=0;
// Setzt den Zähler, welcher die GPS-Daten speichert auf null.
/* Wenn die Bedingung für weitere UART-Messungen zutreffen */
    if(time_catched == true){
        IEC5bits.U4RXIE = 1;
// Dann werden Interrupts von UART-Modul aktiviert
    }
}
```

Die UART_Start-Funktion befindet sich in den Funktionen UART_Read und SD_Data Save. Die Variable a füllt das Array, welches die GPS-Daten empfängt. Bei jedem Aufruf der Funktion wird die Variable a auf null gesetzt, mit der Absicht, dass die neuen GPS Daten die alten überschreiben. Sobald die boolesche Variable time_catched gleich eins ist (true), werden Interrupts von dem UART Modul aktiviert. Dies passiert dann, wenn die Speicherung der Daten komplett ist, oder wenn die aktuelle Zeit der festgelegten nicht entspricht. Die GPS-Datensätze werden nun wieder empfangen und ausgelesen.

4.7.5. SD DataSave

```
void SD_DataSave(){
/* Wenn die Anzahl der Messwerte erreicht ist */
    if(e == anz){

/*Erstellt die Datei wodrauf die Daten gespeichert werden, Datei lautet:
Datum.csv */
        g=0;
        for(c=0;c<6;c++){
            SaveData[c] = gpsdate[g];
            g++;
        }
       g=0;
        for(c=6;c<10;c++){
            SaveData[c] = CSV[g];
            g++;
        }

/* Öffnet die Datei wodrauf die Messung gespeichert wird*/
    f_open(&file, SaveData, FA_OPEN_APPEND | FA_WRITE);
//FA_OPPEN_APPEND = Öffnet die Datei, wenn sie existiert und erstellt sonst eine
neue Datei. Der Schreib pointer wird auf das Ende der Datei gesetzt.
//FA_WRITE = Ermöglicht das Beschreiben der Datei

/* Speichert die Messwerte in der Datei*/
    f_printf(&file, "\n%s\n", gpstime);
    for(g=0; g<e; g++){
    f_printf(&file, "%d\n", E[g]);
    }
/* Schließt die geöffnete Datei wieder */
    f_close(&file);

/* Sobald alles Gespeichert wurden ist, wird die Variable mit der Messwerte
gespeichert werden geleert, damit neue Messwerte da drin gespeichert werden
können. Die Boolesche Variable welche das UART Modul startet wird aktiviert */
    e = 0;
    time_catched = true;
    }
}
```

Die Funktion SD_DataSave befindet sich in der Hauptschleife und wird ausgelöst, sobald die Spannungsmessung vorüber ist. Es folgt die Erstellung der Datei auf der die Spannungsmessung gespeichert werden soll. Es wurde Dateityp CSV ausgewählt, da er es vereinfacht mit Excel die Dateien auszuwerten. Die Funktionen, die zur Speicherung verwendet werden, werden aus der Datei ff.c bezogen. Die Datei ff.c sowie die anderen Dateien, welche ff.c verwenden, stammen von den Programmierern „ChaN“, welche ihre Speicher-Bibliothek im Netz frei zur Verfügung stellen. Durch die Funktion f_open wird die verwendete Dateistruktur definiert. Anschließend wird der Dateiname festgelegt. Im letzten Schritt wird der Modus festgelegt. Der Modus FA_OPEN_APPEND definiert, dass die Datei geöffnet wird, falls sie existiert, ansonsten wird eine neue erstellt. Durch FA_WRITE wird festgelegt, dass die Datei beschrieben werden soll. Die f_print f Funktion schreibt auf die festgelegte Dateistruktur. Es wird zuerst der Zeitpunkt der Messung gespeichert. Anschließend werden mithilfe eine For Schleife die Daten der Spannungsmessung auf der SD-Karte gespeichert. Zum Schluss wird die Datei geschlossen, die Datenabspeicherung ist vorbei. Die Variable e wird auf 0 gesetzt, so dass die Spannungsmessung ihre Dateien überschreibt. Die Boolesche Variable time_catched wird auf wahr(true) gesetzt und startet somit im nächsten Schritt, bei dem Aufruf von UART_Start, die Zeitmessung.

4.8. Programm

Das Programm setzt sich aus dem Hauptprogramm und der darin beinhalteten Endlosschleife zusammen.

4.8.1. Hauptprogramm

```
void main(){
    SYSTEM_Init();
    //Initialisiere Systemeinstellungen - z.B. Taktfrequenz der Peripheren Buse
    INTERRUPT_Init();
    //Initialisiere Interrupts - UART4&ADC0&ADC1
    TIMER_Init();
    //Initialisiere Timer - Zur Taktfrequenz von ADC0&ADC1
    SD_Init();
    //Initialisiere SD-Karte - SPI3
    UART_Init();
    //Initialisiere GPS-Modul - UART4
    ADC_Init();
    //Initialisiere Spannungsmessung - ADC0&ADC1
```

In dem Hauptprogramm werden die Funktionen aufgerufen, welche nur einmal benötigt werden. Dazu gehören die Initialisierungsvorgänge, welche die entsprechende Hardware und Schnittstellen konfiguriert. Es beginnt mit der Initialisierung der Systemeinstellungen, wo die Taktfrequenzen der peripheren Busse und die allgemeinen Register der Interrupts konfiguriert werden. Die nächste Funktion initialisiert die Interrupt-Konfigurationen, dort werden die verwendeten Interrupts entsprechend konfiguriert. Es folgen die Zähler(Timer)-Initialisierungen, hier werden die Abtastfrequenzen der A/D Wandler konfiguriert und gestartet. Als nächstes werden die SPI, UART und ADC Register konfiguriert. Sobald die Initialisierungsvorgänge abgeschlossen sind, wird die Endlosschleife gestartet. Der Initialisierungsvorgang ist nach wenigen Sekunden abgeschlossen, anschließend startet das Programm die Endlosschleife, welche nur unterbrochen werden kann, wenn entweder die SD-Karte voll ist oder die Spannungsversorgung erlöscht.

4.8.2. Endlosschleife

```
/* Endlosschleife - Wird permanent ausgeführt */
    while (1){

// Liest das einkommende GPS Zeitsignal und vergleicht es mit den vordefinierten
Werten.
        UART_Read();

// Speichert die Daten, sobald der Arbeitsspeicher zu 90% gefüllt ist, auf der
SD-Karte.
        SD_DataSave();

    }
}
```

In der Endlosschleife des Hauptprogramms befinden sich die Funktionen welche regelmäßig aufgerufen werden. Dazu gehören die Funktionen UART_Read und SD_DataSave. Diese Funktionen müssen regelmäßig aufgerufen werden, da sie nicht auf ein Interrupt reagieren und es zu viel Prozessorleistung ziehen würde, wenn man sie Interrupt gesteuert aufrufen würde. In den Funktionen sind Bedingungen gestellt, welche mit den Interrupts zusammenhängen, so dass dort schlussendlich sehr oft abgefragt wird, ob die Bedingung zum Start der Funktion erfüllt ist oder nicht.

5. Fazit

Die Aufgabe des Praxisprojekts war die Programmierung eines Phasor-Messgerätes, welches als Vorarbeit zur Bachelorarbeit dienen soll. Die Recherche hat ergeben, dass dies am besten mit einem leistungsstarken Mikroprozessor umgesetzt werden kann. Dabei wurde ein 32-Bit auflösender Prozessor der Firma „microchip“ verwendet, welcher mit einer Taktfrequenz von 200MHz arbeitet. Damit die Phasor-Messgeräte zur selben Uhrzeit die Spannung messen, wurde ein hochpräzises GNSS Modul verwendet, welches die Uhrzeit und das Datum empfängt und maximal um plus/minus ein Millivolt versetzt, das Signal zur Spannungsmessung an den Mikrocontroller weiterleitet. Es wurde untersucht, auf welche Schnittstellen und Module das Phasor-Messgerät zugreifen muss. Zur Ermittlung der Uhrzeit und des Datums kommuniziert das GNSS Modul mit der UART Schnittstelle des Mikrocontrollers. Über die SPI Schnittstelle speichert der Mikrocontroller Daten auf der SD-Karte. Die A/D-Wandler werden benötigt, um die Spannungsmessung aufzunehmen und den Puls-pro-Sekunde, welcher das GNSS Modul liefert. Sobald der Puls-pro-Sekunde gemessen wird und die festgelegte Zeit erreicht wurden ist, startet die Spannungsmessung.
Die Programmierung wurde durch den Zugriff auf die Register des Mikrocontrollers ausgeführt. Als Programmiersprache wurde C verwendet. Zur Hilfestellung wurden Bibliotheken hinzugefügt, welche das Programmieren erleichtern. Die größte Bibliothek beinhaltet Funktionen, um auf Datenspeicher zuzugreifen. Darunter befinden sich die Funktionen, welche den Zugriff auf die SD-Karte ermöglichen. Die Konfiguration der Register beinhaltet die Prozessor-Konfigurationen und die Schnittstellen-Konfigurationen. Während der Prozessor-Konfigurationen wurden System, Variablen, Bibliotheken, Prozessortakt und Interrupt-Einstellungen konfiguriert. Die Schnittstellen-Konfigurationen beinhalten die Initialisierung der Seriellen-Peripheren-Schnittstellen (SPI), der Universellen-Asynchronen-Empfänger-Sender-Schnittstelle (UART), des Analogen-Digitalen-Umsetzer (ADC) und der des Zählers (TIMER). Darüber hinaus verwendet das Programm, um die Schnittstellen auszuwerten, selbstgeschrieben Funktionen. In den Interrupt-Service-Routinen werden GNSS Datensätze, die Puls-pro-Sekunde Flanke und die Spannungsmessungsdaten abgespeichert. Die UART-Read Funktion wertet den GNSS Datensatz aus und speichert die aktuelle Zeit. Falls eine festgelegte Zeit erreicht wurden ist, wird die Funktion Time-Reached ausgeführt und das Interrupt, welches die GNSS Daten einliest deaktiviert. Die Time Reached Funktion startet das Interrupt, welches die Puls-pro-Sekunde Flanke einliest. Sobald das Puls-pro-Sekunde Interrupt auslöst, wird das Interrupt, welches die Daten zu Spannungsmessung einliest aktiviert und das Interrupt, welches die Puls-pro-Sekunde Flanke einliest, wird deaktiviert. Nachdem das Spannungsmessungsinterrupt die Anzahl der vorgegebenen Daten eingespeichert hat, wird das Spannungsmessungsinterrupt deaktiviert. Anschließende wird die Funktion SD-DataSave in der Endlosschleife aktiviert, da dort nun die Bedingung erfüllt ist. Sobald die Funktion die Daten abgespeichert hat, wird die Funktion UART-Start ausgeführt, welche das UART-Interrupt aktiviert. Danach fängt der Prozess wieder von vorne an. Dies geschieht solange, bis das Gerät von Spannungsquelle getrennt wird.
In der Bachelorarbeit soll dann mithilfe eines weiteren Phasor-Messgerätes der Phasenwinkel an zwei verschieden Punkten des Niederspannungsnetzes verglichen und ausgewertet werden. Dabei liegt der Schwerpunkt auf den Nachweis von Leistungsverlusten, welche laut Theorie beim Leistungstransport durch die Leitungen entsteht. Für die Messung muss noch ein Spannungswandler gebaut werden, welcher die Spannung des Netzes auf die Spannung des Mikrocontrollers heruntertransformiert. Es ist geplant die Netzspannung mithilfe von Transformatoren zu dimensionieren. Da es keinen Transformator gibt, welcher die benötigte Spannung des Mikrocontrollers transformiert, wird dahinter mit einem Spannungswandler und einem Gleichspannungsoffset die benötigte Spannung von plus 230V bis minus 230 Volt auf 0 bis 3,3V transformiert und somit der Eingangsspannung des Mikrocontrollers entspricht. Anschließend kann die Messung gestartet und ausgewertet werden.

6. Literaturverzeichnis

[1] Gude Analog- und Digitalsysteme GmbH, „Gude," 2018. [Online]. Available: https://www.gude.info/.

[2] Microchip, „Microchip - Developer Help," 2018. [Online]. Available: http://microchipdeveloper.com/mplabx:requirements.

[3] Microhip, „Datenblatt - PICKit 3," 1 Mai 2010. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/51795B.pdf.

[4] All about circuits, „The Universal Asynchronous Receiver/Transmitter," 20 Dezember 2016. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/back-to-basics-the-universal-asynchronous-receiver-transmitter-uart/.

[5] Wikipedia, „Serial Peripheral Interface," 22 Oktober 2018. [Online]. Available: https://de.wikipedia.org/wiki/Serial_Peripheral_Interface.

[6] Digilent, „Datenblatt - ChipKIT Wi-FIRE," 12 Juli 2016. [Online]. Available: https://reference.digilentinc.com/_media/chipkit_wifire/chipkit_wi-fire_rm_revc.pdf.

[7] Microchip, „Datenblatt - PIC32MZ," 23 Juni 2016. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/60001320E.pdf.

[8] MikroElektronika, „GNSS 5 Click," [Online]. Available: https://www.mikroe.com/gnss-5-click.

[9] Microchip, „Developer Help - Oszillator Konfiguration," 2018. [Online]. Available: http://microchipdeveloper.com/32bit:mz-osc.

[10] ChaN, „FatFs - Generic FAT Filesystem Module," 14 Oktober 2018. [Online]. Available: http://elm-chan.org/fsw/ff/00index_e.html.

[11] A. Mocke, „PIC32 for the hobbyist," 15 Oktober 2018. [Online]. Available: http://aidanmocke.com/.

[12] Microchip, „Datenblatt - A/D Wandler," 14 Juli 2015. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/60001359b.pdf.

[13] Microchip, „Datenblatt - SPI-Schnittsetlle," 02 Februar 2011. [Online]. Available: http://ww1.microchip.com/downloads/en/devicedoc/61106g.pdf.

[14] Microchip, „Datenblatt - UART-Schnittstelle," 29 November 2011. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/61107G.pdf.

Abbildungsverzeichnis