technische universität
dortmund

Lehrstuhl für Kommunikationsnetze
Prof. Dr.–Ing. Christian Wietfeld

Praktikum

Microcontroller
(Modul ETIT–109)

Redaktion:
Dipl.–Ing. Dipl.–Kfm. Ralf Burda

Verantwortlicher Hochschullehrer : Prof. Dr.–Ing. C. Wietfeld
Version vom : 4 March 2022

Inhaltsverzeichnis

Kurzfassung

Das Praktikum „Microcontroller“ führt die Studierenden in die Praxis des Betriebs und der Programmierung von Cyber-Physical-Systems ein, wie sie derzeit in ständig wachsendem Maße in praktisch allen Gerätetypen des täglichen Lebens eingesetzt werden.

Anhand einzelner, aufeinander aufbauender Aufgaben werden z.B. folgende Fähigkeiten trainiert:

- Ableitung einer Lösungsstrategie auf Basis von Anforderungen.
- Umgang mit einer typischen Werkzeugkette zur Programmierung von Microcontrollern.
- Implementierung von einfachen Softwaremodulen in Assembler.
- Implementierung von Softwaremodulen in C/C++.
- Adressierung und Beschreibung der Hardwareressourcen.
- Lesen von Spezifikationsdokumenten zu Hardwarekomponenten.
- Lesen und Verstehen von Schaltplänen.

Die in dieser Beschreibung in Kapitel 3 vorgestellten Aufgabenstellungen sind zunächst konsekutiv zu bearbeiten, um die Grundfunktionen der MCU und den grundsätzlichen Umgang mit den Hardware-Ressourcen zu erlernen. Die Aufgaben aus Kapitel 4 dienen der Vertiefung der erlernten Fähigkeiten. In der Regel sollten 3 der 6 Aufgaben gelöst werden. Je nach Vorkenntnissen oder Lernkurve sind auch alle Teilexperimente in der Zeit des Praktikums durchführbar.

Sicherheitshinweise

Um Probleme beim Betrieb und Schäden an den Geräten durch unsachgemäße Handhabung zu vermeiden, beachten Sie bitte folgendes

- Sorgen Sie dafür, dass auf dem Breakout–Board keine Kurzschlüsse zwischen den Kontakten entstehen
- Bitte schließen Sie das beiliegende Steckernetzteil nur dann an, wenn Ihre Software die LED–Leiste und/oder die Motoransteuerung betreibt!
- Drücken Sie niemals den „Reset“ Knopf, während das Programmiergerät angeschlossen ist!

1 Einleitung

1.1 Die Experimentierplattform

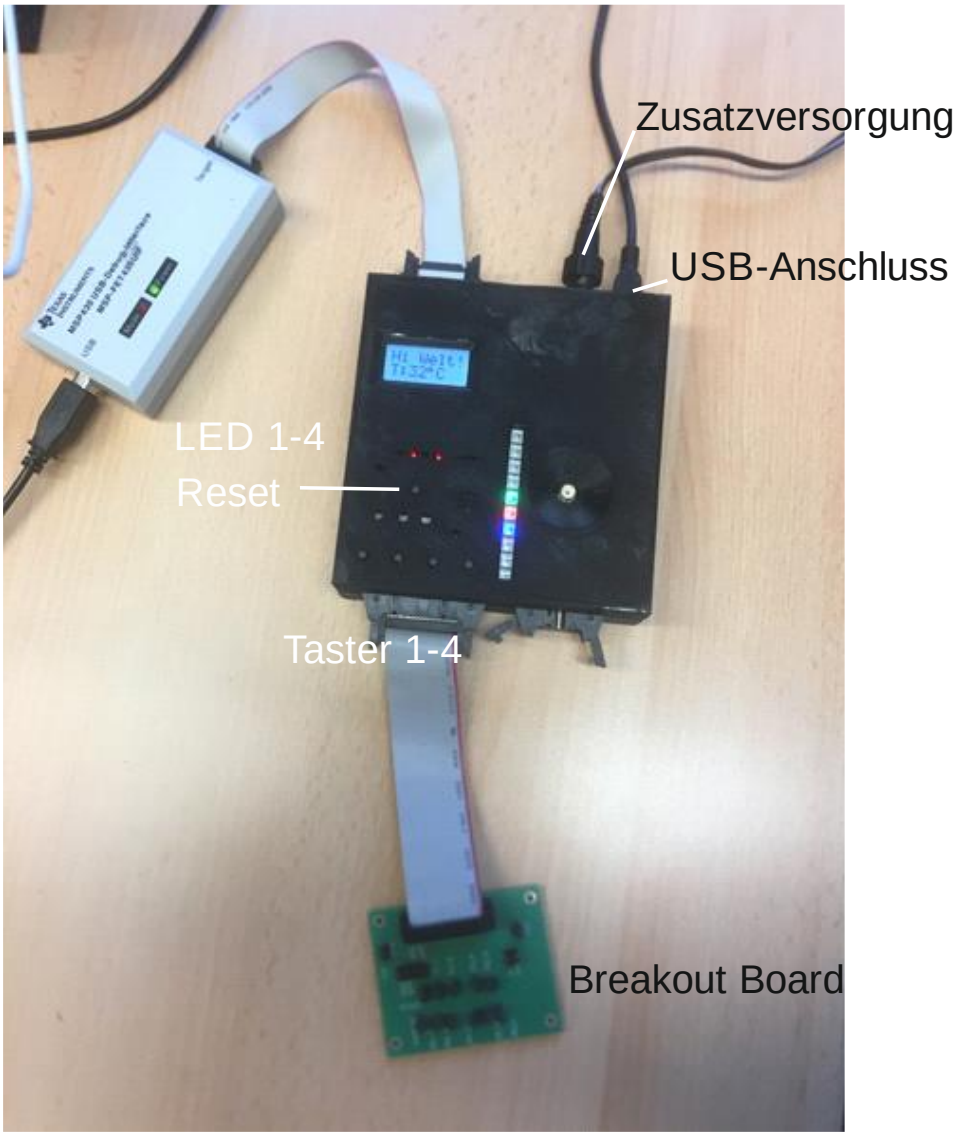

Abbildung 1: Vollständiger Aufbau der Fallstudie

Auf dem Experimentierboard sind, entsprechend der im Moodle verfügbaren Schaltpläne in „Basisboard.pdf“, ein Mikrocontroller des Typs MSP430F2619 sowie die notwendigen externen Anschlüsse und eine Spannungsversorgung implementiert. Für die Ein– und Ausgabe stehen ein Tastenfeld mit LEDs (Upperboard.pdf) sowie ein LCD–Display und eine serielle Schnittstelle zum PC zur Verfügung. Letztere wird über das USB Interface abgebildet. Die USB Schnittstelle dient auch gleichzeitig zur Spannungsversorgung des Mikrocontrollers und der meisten Elemente auf dem Grundgerät.

In den Aufgabenteilen, in denen Motoren oder Lüfter sowie die Reihe der Leuchtdioden unter dem Sichtfenster betreiben werden, (und nur hierbei) benötigen Sie noch das zusätzliche Netzteil

Um weitere Komponenten für Experimente anschließen zu können dient das „Breakout Board“. Auf diesem sind einzelne Anschlusspins bzw. ganze Schnittstellen durch die Farben nun Formen der Steckverbinder codiert.

In diesem Versuch kommen zum Einsatz:

1) Ein Standlüfter zur Veranschaulichung einer einfachen Motorsteuerung, anzuschließen an den USB-Anschluss des Breakout-Boards.
2) Ein Axiallüfter mit Drehzahlausgang zur temperaturabhängigen Regelung. (blauer Stecker)
3) Ein Servomotor (je nach Bauform schwarzer oder weisser Stecker am Servo, weisse Buchse auf dem Breakout-Board)
4) Ein DCF–77 Empfangsmodul zur Implementierung einer Funkuhr. (Bordeaux-farbener Stecker)
5) Ein Motor/Lüfter mit Vorwärts/Rückwärtslauf (schwarzer, 2-poliger Stecker)

1.2 Didaktische Zielsetzung des Praktikums

Es werden Wissen und Fähigkeiten vermittelt, wie automatisierte Umgebungen die Qualität des Softwareentwicklungsprozesses für Kommunikationssysteme positiv beeinflussen und die Arbeitsabläufe effizient gestalten. In diesem Kontext werden auch Methoden und Kompetenzen in der Teamarbeit vermittelt.

Operativ werden diese Fähigkeiten bei der Umsetzung des Systems der Fallstudie trainiert und die notwendigen, einzelnen Arbeitsschritte bis zur erfolgreichen Umsetzung einer Lösung erprobt.

Alle erfolgreichen Teilnehmer sollen am Ende des Praktikums Fähigkeiten entwickelt haben, um die folgenden Aufgaben übernehmen zu können:

- Identifikation von Teilproblemen bei der Lösung einer gestellten Entwicklungsaufgabe.
- Bewertung von Werkzeugen im Hinblick auf die Nutzbarkeit für bestimmte Problemklassen.
- Spezifikation von Arbeitsabläufen durch Wahl geeigneter Methoden.
- Erstellung von einfachen Anforderungsdokumenten.
- Erstellung von einfachen Prüfplänen.

1.3 Zeitplanung

Das Praktikum besteht aus insgesamt 10 Sitzungen zu je 5 (Zeit–)Stunden. Eine weitere Sitzung kann bei Bedarf, z.B. zur Vervollständigung von Dokumenten, in Absprache mit der Praktikumsleitung als Ersatz für das notwendige Eigenstudium, eingerichtet werden.

Für Studierende aus dem B.Sc. Wirtschaftsingenieurwesen ist eine Verlängerung auf 12 Sitzung (4ECTS) oder 15 Sitzung (5 ECTS) bedarfsgerecht möglich.

Die einzelnen Sitzungen werden wechselnd für die Einführung in Werkzeuge und Arbeitstechniken und für die Durchführung praktischer Tätigkeiten der Studierenden genutzt.

Die Inhalte der Sitzungen bauen aufeinander auf, so dass eine kontinuierliche Mitarbeit der Studierenden notwendig ist.

Der zeitliche Umfang lässt es in Absprache mit der Praktikumsleitung in der Regel zu, dass parallele Pflichtveranstaltungen wie Übungen oder Prüfungen im Umfang von max. 6 Zeitstunden wahrgenommen werden können.

1.4 Berichte

Zum Abschluss des Praktikums ist ein Praktikumsbericht einzureichen. Dieser soll für jede Aufgabe folgende Inhalte enthalten:

1) eine Skizze/Paraphrasierung der Problemstellung
2) Erläuterung und Begründung des eigenen Lösungskonzeptes
3) die Darstellung der Lösung
4) Beschreibung der Testprozedur, um die gewünschte Funktion zur Problemlösung sicherzustellen

Insbesondere bei der Testprozedur ist gegebenenfalls darauf zu achten, die Eingaben und deren zeitliche Abfolge so zu variieren, dass alle Teile der Software auch tatsächlich durchlaufen werden.

1.5 Aufgabenstellungen

Die Experimentieraufgaben sind in Fließtext eingebettet, bei dem sich Erläuterungen, Problemstellungen und Arbeitsanweisungen abwechseln. Die Stellen, an denen von den teilnehmenden Personen Eigenleistung erforderlich ist, sind durch 2 Symbole am Seitenrand gekennzeichnet:

an dieser Stelle finden Sie im Moodle einen kleines Quizz, mit dem Sie Ihr Verständnis der vorangegangenen Erläuterungen testen können.

An dieser Stelle gibt es etwas zu tun. In der Regel sind Einstellungen an der Hardware vorzunehmen bzw. zu überprüfen oder die Software zu ergänzen.

2 Die Arbeitsumgebung

2.1 Verwendete Werkzeuge

2.1.1 Das Code–Composer–Studio (Eclipse)

Das Code–Composer–Studio (CCS) ist eine modifizierte Eclipse Umgebung, die auf die Entwicklung von Code für Prozessoren der Fa. Texas–Instruments spezialisiert ist. Detaillierte Informationen zu den einzelnen Arbeitsabläufen finden Sie in der Beschreibung der Aufgaben. Sämtliche Entwicklung, auch das Debugging und die Darstellung interner Zustände des Micro–Controllers werden über diese Anwendungsoberfläche vorgenommen.

2.1.2 Der Anschluss der Experimentierbox

Die Experimentierbox wird über eine Mini–USB Kabel an den PC angeschlossen. Die notwendigen Treiber sind vorinstalliert. Gehen Sie wie folgt vor:

a. Schließen Sie das beiliegende Mini–USB Kabel an den PC und die Experimentierbox an.
b. Das Display sollte nach kurzer Zeit aufleuchten.
c. 4 rote Leuchtdioden sollten leuchten.

ACHTUNG: Dieses Setup wird vor Beginn des Praktikums voreingestellt und wird durch Ihre Programmieraktivitäten überschrieben. Es dient nur zur Sicherstellung, dass Ihr Arbeitsplatz korrekt funktioniert!

2.1.3 Die Debugger/Programmier–Schnittstelle.

Über ein 14–pol. Flachbandkabel wird das MSP430 USB–Debug–Interface an die Experimentierbox angeschlossen. Zur Programmierung und zum Debugging wird hier ein so genanntes JTAG–Interface implementiert und per USB Verbindung aus dem Code–Composer–Studio betrieben. Je nach Ausstattung Ihrer Experimentierbox ist dies ein MSP-FET430UIF (grau) oder ein MSP-FET (schwarz). Die beiden Geräte unterscheiden sich in Ihren genutzten Funktionen nicht, so dass im weiteren keine Unterscheidung gemacht wird.

Die Debugger–Schnittstelle sollte gemäß Abbildung 1 durchgehend verbunden sein. Wird die Schnittstelle vom PC korrekt konfiguriert, so leuchtet auf dem Adapter die grüne LED.

Wird nach dem erfolgreichen Erstellen einer Software mittels CCS das Symbol „Debug“ in der Oberfläche geklickt, so lädt die Entwicklungsumgebung den Binärcode in den Microcontroller und stoppt die Ausführung bei der ersten Instruktion (dem Programmstart) des Quelltextes. Für Assembler–Programme ist dies die Lokation des Reset–Einsprungs, für C–basierten Code der Eintritt zur Routine „main“.

3 Konsekutive Aufgaben

3.1 Einarbeitung in die Arbeitsumgebung

3.1.1 Erste Schritte

In dieser Aufgabe werden Sie die Entwicklungsumgebung und die Bedienung des Programmiergerätes sowie erste Debugging–Möglichkeiten kennenlernen.

1. Verbinden Sie das Entwicklungsboard korrekt mit Ihrem Computer, so wie es in Abbildung 1 dargestellt ist.
2. Starten Sie das Code–Composer Studio.
3. Erstellen Sie ein neues Projekt (File→New→CCS–Projekt)
    a. Wählen Sie ein „Empty–Assembler–Only Project“
    b. Wählen Sie als Target „MSP430x2xx Family
    c. Wählen Sie die MCU MCP430F2619
    (**Hinweis**: die Einstellungen für die MCU müssen Sie bei jedem der neuen Projekte erneut vornehmen)
    d. Geben Sie als Projektnamen „Aufgabe0“ an und klicken Sie danach auf „Finish“.

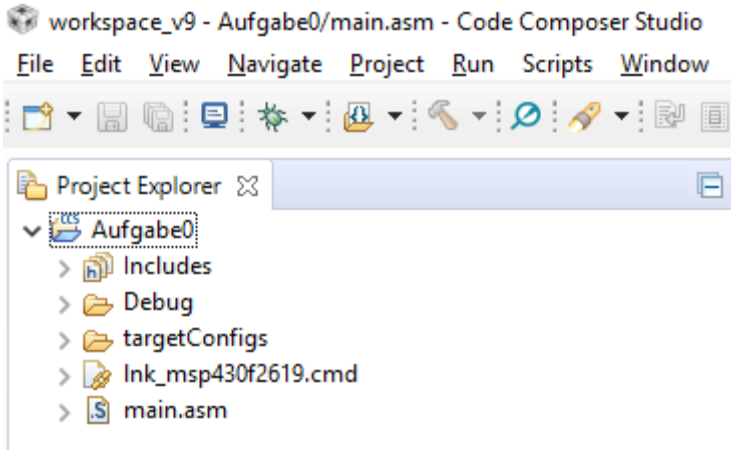

Abbildung 2: Erstes Projekt

An dieser Stelle sollte Ihre Arbeitsplatzansicht wie in Abbildung 2 dargestellt erscheinen.

Die Konfiguration der weiteren Arbeitsumgebung ist hiermit abgeschlossen.

4. Verbinden Sie nun die Experimentierbox (im Folgenden kurz „Box“ genannt) per USB mit Ihrem Computer. Dabei sollten die Sie folgenden Geräte und Kabel verwenden:
    a. Experimentierbox
    b. Programmieradapter MSP430–FETUIF/MSP-FET
    c. USB Kabel A/B zur Verbindung des Programmieradapters

d. USB Kabel A/mini–B zur Verbindung der Experimentierbox

Im Ausganszustand sollte jetzt auf dem Programmieradapter eine grüne LED leuchten. Des Weiteren zeigt das Display der Box den Text „Hi Welt!“ und die aktuelle Temperatur an. Darunter sollten 4 rote Leuchtdioden rhythmisch blinken.

ACHTUNG: Diese Testsoftware wird mit Ihrem ersten Programmierversuch überschrieben und in exakt dieser Form während des Praktikums nur einmal wiederhergestellt.

HINWEIS: Erstellen Sie für jede weitere Aufgabe ein neues Projekt. Da die Programme aufeinander aufbauen, kopieren Sie bitte immer zu Beginn einer neuen Aufgabe alle Quelldaten des vorherigen Projektes in das neue Projekt der jeweiligen Aufgabe.

5. Fügen Sie hinter dem Kommentar „main loop“ den Befehl „jmp $“ ein und wählen Sie danach im Menu „Project→Build Project“.Alternativ klicken Sie auf den Hammer im Hauptmenu. (siehe Abbildung 3.) Nach einiger Zeit (i.d.R. werden im Hintergrund noch einige Bibliotheken für Ihr Programmierziel erstellt) erscheint die Nachricht „**** Build Finished ****“.

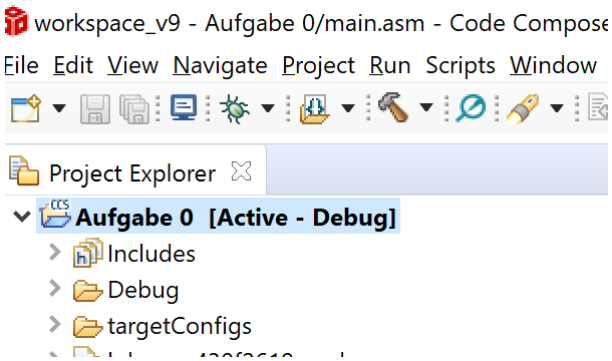

Abbildung 3: Hauptmenu der Arbeitsoberfläche

Jetzt ist Ihr Projekt bereit für die Programmierung, d.h. die Übertragung in den nicht flüchtigen Speicher des Mikrocontrollers und das anschließende Debugging bzw. die Nutzung des erstellten Programms.

6. Starten Sie nun das Debugging durch Klick auf das Debugging Symbol.
   Es erscheint eine Messagebox, die sich nach kurzer Zeit wieder schließt und in einem Bildschirm der, wie in Abbildung 4 gezeigt, die Debug–Ansicht darstellt. Die Ausführung der CPU ist gestoppt. Der nächste, auszuführende Befehl, ist farblich unterlegt und am linken Bildschirmrand zeigt ein dreieckiger Cursor auf eben diese Zeile.

```
10                                          ; make it known to linker.
11 ;-------------------------------------------------------------------------
12             .text                        ; Assemble into program memory.
13             .retain                      ; Override ELF conditional linking
14                                          ; and retain current section.
15             .retainrefs                  ; And retain any sections that have
16                                          ; references to current section.
17
18 ;-------------------------------------------------------------------------
19 RESET       mov.w   #__STACK_END,SP      ; Initialize stackpointer
20 StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
21
22
23 ;-------------------------------------------------------------------------
24 ; Main loop here
25 ;-------------------------------------------------------------------------
26     jmp $
27
28
```

Abbildung 4: Ansicht des gestarteten Debuggers

Grundlagen der Programmierung und der Denkmodelle, denen die Funktion der MCU folgt, sind in den nächsten Abschnitten beschrieben. Eine detaillierte Einführung in Grundlagen der Programmierung, Denkmuster und Nomenklaturen finden Sie in „Anhang A, Allgemeines zur Programmierung“.

3.1.2 Erste Debugging Schritte

Zur Ausführung Ihres ersten Programms (und aller Weiteren in diesem Versuch) gibt es nun unterschiedliche Möglichkeiten. Alle diese Optionen finden Sie in der Symbolleiste der Debug–Ansicht. Fahren Sie mit der Maus über die Symbole, um Schnellhilfen zu erhalten.

1) Programm starten und (frei) laufen lassen.
2) Einzelschritt ausführen
3) Programmablauf pausieren
4) Programm zum Start zurücksetzen (Reset)
5) Programmablauf stoppen und Debugging beenden.

Bitte beachten Sie, dass nach dem Stoppen des Programms die Software neu in den Flash–Speicher geladen wird. Da dieser Vorgang auf die Dauer die CPU zerstört (nur ca. 10.000 Brennvorgänge sind üblicherweise spezifiziert), nutzen Sie für einen weiteren Test derselben Software bitte die „Reset“ Funktion.

Durch einen Doppelklick links in der Spalte neben dem Quelltextfenster können Sie einen Breakpoint setzen. Wird die so markierte Stelle im Quelltext erreicht, so stoppt die Ausführung des Programms und die Debug–Oberfläche wird wieder aktiv.

Setzen Sie nun die Software zurück, markieren Sie die zweite Zeile (StopWDT….) mit einem Breakpoint und lassen Sie nun die Software frei laufen. Sie werden sehen, dass die Ausführung stoppt.

AKTION: Wenn Sie die Funktionen gefunden haben, lösen Sie den entsprechenden Test im Moodle!

3.1.3 Allgemeines zur Programmierumgebung

3.1.3.1 Spezifika der MSP430

In diesem Praktikum wird der MSP430 Mikrocontroller zunächst direkt mittels Assemblerbefehlen programmiert. Im Vergleich zu Hochsprachen wie Java oder C stehen Ihnen in Assembler viele vertraute Programmiertechniken nicht zur Verfügung, können aber mit Assemblerbefehlen nachgebaut werden. Machen Sie sich mit dem Befehlssatz des MSP430 vertraut (siehe Befehlsübersicht und das Dokument „MSP430 Assembly Tool“ im Moodle). Lernen Sie die Ihnen zur Verfügung stehenden Befehle und ihre Bedeutung kennen. Achten Sie auf folgende Besonderheiten:

- Die Register des MSP430 sind in der Regel 16 Bit breit. Es gibt jedoch auch zahlreiche 8–Bit Register. Um diese als Zieloperand zu verwenden, muss der entsprechenden Instruktion ein '**.B**' angehängt werden.
- Der MSP430 unterscheidet allgemeine Register und so genannte „**S**pecial **F**unction **R**egister“ (SFR). Verwenden Sie für diese das falsche Format, erscheint in der Regel die folgende Fehlermeldung

  „$Accessing SFR using incorrect size$

- Möchten Sie einen konstanten Wert als Operanden verwenden, so müssen Sie diese mit einer vorstehenden Raute '#' kennzeichnen. Eine Zahl kann in verschiedenen Darstellungsarten verwendet werden: dezimal (**#19**), hexadezimal (**#0x13**) oder binär (**#1011b**).
- Stellt der Operand eine Speicheradresse dar, so muss der Operand ‚&‘ vorangestellt werden, z.B.

  ```
  mov.b #0x0F, &0x0002
  ```

  In diesem Falle würde der Wert „0x0F“ an die Adresse „0x0002“ im Speicher der MPU geschrieben.
- Grundsätzlich gilt, dass alle Registernamen, wie sie im Family–User Guide beschrieben sind, von der Software als Adressen aufgefasst werden. Alle Bitmasken zum Ansprechen einzelner Funktionen innerhalb eines SFR hingegen werden als nicht näher definierte Zahlen interpretiert und bedürfen daher bei der Nutzung das „#“ Präfix. So sind die Notationen

  ```
  Mov.b #0x0F, P1DIR
  ```

  Und

  ```
  Mov.b #BIT3 + BIT2 + BIT1 + BIT0, P1DIR
  ```

  äquivalent. Bei vielen SFRs sind die Namen der einzelnen Bits als Funktionsnamen hinterlegt. In allen Fällen bietet sich auf Grund der besseren Lesbarkeit des Quelltextes immer die Nutzung von Symbolen an Stelle von hart–codierten Zahlenwerten an.
- Während die Funktion „mov“ immer das ganze Wort (oder mov.b ein Byte) beschreibt, erlauben andere Funktionen die Manipulation einzelner Bits.
  - **BIS**, bit set, setzt alle die Bits innerhalb des Ziels auf ‚1‘, die Operanden angegeben sind. Dies entspricht einer logischen „Oder“ Operation mit einer Bitmaske.

- **BIC**, bit clear, löscht alle die Bits innerhalb des Ziels auf ‚0', die Operanden angegeben sind. Dies entspricht einer logischen „Nicht–UND" Operation mit einer Bitmaske.
- **BIT**, bit test, prüft, ob mindestens ein Bit der als Operand gegebenen Maske im betroffenen Ziel gesetzt ist. Der Inhalt des Ziels wird dabei nicht verändert.

Für weitere programmierrelevante Informationen konsultieren Sie bitte den „Family User Guide" sowie das „MSP430 Instruction Set Summary". Für Details der verfügbaren Funktionen und Module im MSP430F2619 schauen sie gleichermaßen in das entsprechende Data–Sheet. Die entsprechenden Anweisungen in C/C++ sind in Anhang A.6 Manipulation von Werten, dargestellt.

3.1.4 Kontrollstrukturen in Assembler–Code

Im Vergleich zu höheren Programmiersprachen, in denen unterschiedliche Kontrollkonstrukte wie „for", „while" oder „if" vohanden sind, bietet die Assembler–Programmierung nur eingeschränkte Formulierungen für die Definition von Kontrollanweisungen. Nach einer Kontrollanweisung verzweigt die Ausführung entweder auf den nächsten Befehl oder auf die als Operand angegebene Adresse.

- Zieladressen werden immer als so genannte Labels angegeben. Ein Label wird dabei durch einen in der ersten Spalte einer Textzeile beginnenden Namen deklariert. (Siehe „RESET" und „StopWDT" im ersten Beispiel.)
- **Jmp label** ein unbedingter Sprung, der immer auf das als Ziel genannte Label als nächster Instruktion führt.
- **JE label** ein bedingter Sprung, hier „**J**ump if **E**qual". Dieser (oder allen ähnlichen Jxx) Instruktion muss immer eine arithmetische Operation vorausgehen, die prozessorinterne Flag–Bits modifiziert. Abhängig davon, ob die geprüfte Bedingung wahr ist, verzweigt der Programmablauf zum genannten Label oder es wird einfach der nächste Befehl ausgeführt.
- **Call #label** ein Aufruf einer Prozedur. Diese ähnelt dem „Jmp", legt aber auf dem Stack die Adresse des nächsten Befehls nach der Call Instruktion ab. Wichtig ist hierbei, dass die aufgerufene Routine IMMER mit dem Befehl „ret" beendet wird, um zu eben dieser nächsten Instruktion zurückzukehren.

ACHTUNG: vergessen Sie das ‚#' nicht. Andernfalls interpretiert der Assembler den Wert des Labels als Speicheradresse, von der die Adresse der nächsten auszuführenden Instruktion geladen wird. Dies nennt man auch einen „indirekten Aufruf (Call)".

3.1.5 Kontrollstrukturen in Assembler, Aufgabe 4.2

Gegeben sei das C–Fragment

```
int i;

int k = 0;

for ( i = 1; i < 10; i++) { k = k+i; }
```

a) Formen Sie dieses Fragment so um, dass es bei gleicher Funktion ausschließlich mit der Entscheidung „if" funktioniert und Labels und Goto

Statements nutzt. Nutzen Sie nun die Befehle des MSP430, um eben dieses umgeformte Fragment in Assembler zu codieren.

b) Debuggen Sie Ihr Programm und verifizieren Sie die Funktion durch Betrachtung der Speicherstellen, in denen Sie i und k gespeichert haben.

Geben Sie eine mögliche Lösung in den entsprechenden Moodle Test ein. Nach dem erfolgreichen Absolvieren geht es weiter!

3.2 Ansteuerung von externen Ports

3.2.1 Arbeit mit Schaltplänen und Handbüchern.

In diesem Abschnitt sollen Sie aus den Schaltplänen der Box die zur Verfügung stehenden Ports ermitteln und für die gewünschte Funktion an der MCU konfigurieren. Hierzu sollten Sie

- Im Schaltplan „Basisboard“ die relevanten Ausgänge suchen, die zu den LEDs führen.
- im MSP430…Datasheet am Seite 77 die Schaltpläne für die relevanten Ports studieren
- sich im MSP430… Family User Guide über die Konfiguration der Register für die jeweiligen Pins informieren.

HINWEIS: Nach erfolgreicher Bearbeitung dieses Schrittes wird für alle weiteren Aufgaben eine komplette Belegungsübersicht im Moodle freigeschaltet.

Füllen sie für sich die folgende Tabelle aus:

Bauteil	Port	Bit
LED 1		
LED 2		
LED 3		
LED 4		
Taster 1		
Taster 2		
Taster 3		
Taster 4		

- Welchen Spannungspegel muss der Ausgang führen, wenn die LED leuchten soll? Welchen binären Wert muss der Ausgang also für eine leuchtende LED haben, wenn eine „1“ als Versorgungsspannung und eine „0“ als Masse (0V) dargestellt wird?
- Die Taster sind so beschaltet, dass ein Tastendruck den Eingang mit der Schaltungsmasse verbindet. Im Ruhezustand soll ein interner Pull–Up Widerstand einen „1“ Pegel garantieren. Beachten Sie die notwendige, passende Konfiguration der entsprechenden Port–Bits.

3.2.2 Experimente zu den LEDs und den Tastern

Starten Sie nun mit einer der Kopie des Projektes aus der ersten Aufgabe. Starten Sie das Debugging.

Öffnen Sie über das Menu Window→Show View→Register die Ansicht der MCU Register.

- Setzen sie geplanten Bits in den einzelnen Registern. Dazu klicken Sie in das Wertefeld und geben dann den neuen Wert ein. Bestätigen Sie die Eingaben mit der Enter-Taste. Die Änderungen sind sowohl für das ganze Register als

auch gezielt für einzelne Bits möglich. Dazu lässt sich die Registeranzeige baumartig bis auf die Bitebene ausklappen.

- Beobachten Sie an der Hardware und protokollieren Sie, ob die gewünschten Effekte so wie geplant auftreten.

HINWEIS: Es kann vorkommen, dass nach einer Eingabe zwar eine Reaktion der Hardware sichtbar wird, aber diese Änderung sich in der Registeranzeige nicht sofort darstellt. Bitte gehen Sie dann im Einzelschrittmodus einen Schritt vorwärts. Dadurch wird die Anzeige immer mit dem vollständigen internen Status der MCU synchronisiert. Insbesondere bei Eingaben an den Tastern ist dies wichtig, da sonst der neue Wert am Eingang zunächst nicht dargestellt wird

ACHTUNG: Nach dem ersten Programmieren der MCU werden weder das LCD–Display noch die Leuchtdioden funktionieren, denn Ihre Programmierung hat das Testprogramm, das zu Beginn des Praktikums geladen war, gelöscht!

Gehen Sie mit Ihrem Lösungsvorschlag ins Moodle und führen Sie den Test zum Thema „Portansteuerung und LEDs“ durch.

3.2.3 Erstellung von Initialisierungsroutinen und Test

Schreiben Sie eine Methode, die die I/O Ports für Taster und LEDs passend konfiguriert. Rufen Sie diese Methode aus dem Hauptprogramm direkt nach der Initialisierung auf.

Schreiben Sie eine weitere Assembler Methode, mit der der aktuelle Zustand der Taster geprüft werden kann.

Rufen Sie diese Methode in einer Endlosschleife auf und lassen Sie die entsprechende LED immer dann leuchten, wenn der Taster gedrückt ist.

3.2.4 Übergang zu C

Erstellen Sie nun ein C–Projekt und codieren Sie die soeben in Assembler erfolgreich implementierten Funktionen in C.

HINWEIS: die symbolischen Konstanten für Register und deren Bits funktionieren in C analog. Sie werden über die Datei <msp430.h> eingebunden und während des Übersetzungsvorgangs in die korrekten Bitwerte/Zahlenwerte übersetzt.

3.2.5 Aufbau der Initialisierung in C

Schreiben Sie zusätzlich eine Methode, in der sie der Reihe nach in die I/O Initialisierung und weitere Initialisierungen aus den kommenden Aufgaben aufrufen werden.

Dies übergeordnete Initialisierungsmethode sollte direkt am Anfang der „main“ Routine, aber nach dem Abschalten des Watchdogs, aufgerufen werden.

Testen Sie Ihre Software abschließend erneut. Ist es auch möglich, beliebige Tasten parallel zu drücken? Prüfen Sie jetzt Ihre Tastenfunktion durch die folgenden Abläufe:

1. Immer, wenn eine Taste gedrückt gehalten wird, soll die entsprechende rote LED auf der Anzeige Leuchten.
2. Mit einem einmaligen Drücken der Taste soll der Status der entsprechenden LED wechseln (aus–an–aus usw.)
3. Die Umschaltfunktion soll mit allen Tasten, auch gleichzeitig gedrückt, getestet werden.

Sie werden in den Punkten 2 & 3 zeitweise auf Probleme stoßen, d.h. Ihre Software wird wahrscheinlich anscheinend manchmal funktionieren und manchmal nicht! Dies ist in diesem Moment normal. Beschreiben Sie das Problem und überlegen Sie woran es liegen könnte.

Ein Tipp: Sie sind hier an einer Schnittstelle zwischen realer Welt und deren digitalem Abbild unterwegs. Welche Bedingung müssen Sie bei Digitalisierung von Informationen immer einhalten, damit keine Fehlinterpretationen auftreten?

Weitere Hinweise erhalten Sie, wenn die Problembeschreibung schriftlich fixiert wurde.

3.3 Konfiguration der Takte und Nutzung von Timern

3.3.1 Beschreibung des Taktsystems der MSP430 MCU

Die verwendete MCU hat ein komplexes Taktsystem, das 3 unterschiedliche Takte für den Betrieb der CPU und der peripheren Einheiten zur Verfügung stellt. Das Taktmodul ist in Kapitel 5 des „Family User Guides“ im Detail beschrieben.

HINWEIS: Hier, wie auch in den weiteren Aufgaben gilt, dass ein konsekutives und vollständiges Lesen eines Handbuches zunächst nur verwirrt. Suchen Sie immer nach den in den Fragen verwendeten Stichworten und lesen Sie selektiv und zielführend. Es gilt immer, dass allgemeine Informationen vor den Details dargestellt werden.

3.3.2 Einarbeitung

Lesen Sie das Kapitel 5 und beantworten Sie (als Vorgriff auf Ihren späteren Bericht) zunächst die folgenden Fragen:

a) Welche Takte stehen innerhalb der CPU für welche Zwecke zur Verfügung?
b) Welcher Oszillator wird nach einem Reset zunächst ausschließlich für die Erzeugung der Takte verwendet?

Bei manchen Operationen wird es später notwendig sein, mit Takten zu arbeiten die zwar unterschiedliche Frequenzen, aber konstante Phasenlagen haben. Werden Takte verwendet, die von unterschiedlichen Oszillatoren stammen, so kann es unter bestimmten Umständen zu Problemen kommen.

c) Suchen Sie, auch in Wikipedia, nach den Begriffen „Deglitching“ und „Race–Conditions“. Beschreiben Sie in Ihrer Ausarbeitung, was dies bedeutet und diskutieren Sie Ihr Ergebnis im Team oder mit dem Versuchsleiter.

3.3.3 Einstellung der Takte

Nutzen Sie die Code–Beispiele aus dem Kapitel 5 um die Takte wie folgt zu konfigurieren:

- MCLK = 16Mhz
- ACLK = 2MHz
- SMCLK = DCOCLK = 12MHz

Konfigurieren Sie nun die relevanten Pins am Microcontroller (siehe Schaltplan „Basisboard.pdf“) so, dass alle 3 Takte auf die gleichnamigen Pins des Breakout–Boards ausgegeben werden und kontrollieren Sie Ihr Ergebnis mit dem Oszilloskop.

Für alle weiteren Versuchsteile können Sie nun annehmen, dass die CPU mit der Frequenz von 16MHz getaktet wird, sofern sie nicht später in einem Energiesparmodus schläft.[1]

[1] Hinweis: Der Code zur Initialisierung der Timer steht praktisch bereits im Handbuch. Allerdings in Assembler, so dass Sie nun Ihre Fähigkeiten aus den vorigen Versuchsteilen nutzen können, um zwischen Assembler und „C“ die notwendige Umformulierung zu leisten. Achten Sie vor allem darauf, dass Sie (siehe auch A.6 Manipulation von Werten) teilweise nur einzelne Bits setzen oder löschen dürfen, um bereits getätigte Einstellungen nicht wieder zu überschreiben.

3.3.4 Verwendung von Timer A0

Der Timer A0 (wie andere Timer der MCU auch) ist eine Hardwareeinheit, mit der Zählereignisse (Takte) gezählt und auf bestimmte Zählerereignisse bzw. Zählerstände reagieren kann. Lesen Sie das Kapitel 12 des Family User Guides und stellen Sie Timer A0 so ein, dass er alle 10ms ein Ereignis erzeugt.

- Wählen Sie eine geeignete Frequenz unter den verfügbaren Frequenzen aus. Beachten Sie, dass der Zähler bis maximal 65535 (0xFFFF) zählen kann.
- Nutzen Sie den „UP“ Mode.
- Stellen Sie eine Zählerschwelle ein (Compare–Register), die 10ms nach dem Start des Timers erreicht wird.

3.3.5 Verwendung von Interrupts zur Reaktion auf Zählerereignisse

Für diese Aufgabe müssen Sie zunächst identifizieren, mit welchem Interrupt der Timer auf das Erreichen der Zählschwelle reagiert.

Schreiben Sie nun eine Interrupt–Service Routine, die immer dann aufgerufen werden soll, wenn der Timer sein Zählereignis (hier Überlauf) liefert. Eine solche Routine muss immer in der Form

```
#pragma vector=TIMER0_A0_VECTOR
__interrupt void
ta0cc0_isr (void) {}
```

angegeben werden. Der Name der Routine ist dabei unerheblich. Die Zuordnung wird aus dem Kontext der „#pragma“–Anweisung korrekt erkannt. Die Namen der „Interrupt–Vektoren“, also die Liste der benannten Einsprungspunkte für die Behandlungs–Routinen, finden Sie auf Seite 15 des Datasheets der MCU.

Innerhalb des Interrupt–Handler implementieren Sie nun die Aktionen, die Sie als Reaktion auf das Ereignis ausführen wollen. In der Regel werden Sie die Interrupt–Flags der jeweiligen Einheit auswerten müssen. Achten Sie daher auf die Beschreibung der einzelnen Register am Ende von Kapitel 12!

3.3.6 Blinken einer LED

Implementieren Sie mit Hilfe einer globalen Variablen das Blinken einer LED so, dass diese pro Sekunde genau einmal blinkt.

3.3.7 Implementierung eines Binärzählers

Nutzen Sie die LEDs 1–4 (LED 4 steht rechts!), um die Werte eines binären 4–Bit Zählers darzustellen.

3.3.8 Erkennung unterschiedlicher Tastenaktivität

3.3.8.1 Einführung zu Tastaturereignissen

In den beiden vorangehenden Aufgabenteilen haben wir gelernt, zeitgesteuerte Aktionen auszuführen. In dieser Aufgabe wollen wir diese neue Fähigkeit nutzen, die Probleme der Aufgabe 3.2.5, namentlich das Prellen der Taster, durch Abtastung zu lösen.

HINWEIS: Unter Abtastung wollen wir die, hier regelmäßige, Erfassung eines Messwertes zu einem bestimmten Zeitpunkt verstehen.

Bei der Anwendung von PCs sind wir es gewohnt, dass nicht nur der Tastendruck selbst, sondern auch verschiedene Variationen möglich sind. Von der Maus kennen wir z.B. den Doppelklick, von der Tastatur die Repeat–Funktion beim langen Drücken einer Taste. Um den Komfort und die Möglichkeiten der Eingabe in den weiteren Aufgabenteilen zu erhöhen, sollen nur die Tasteneingaben mittels des Timers überwacht und unterschiedlichen Ereignissen zugeordnet werden.

In diesem Aufgabenteil sollen die einzelnen Tasten so abgefragt werden, dass, ähnlich wie bei einer PC–Maus oder einer Tastatur, neben der einzelnen Taste auch Ereignisse wie:

- Einzelklick
- Doppelklick
- Dauer–Halten

erkannt werden können. Dazu soll dann noch die Möglichkeit eröffnet werden, die entsprechende Auswertung auch für Tastenkombinationen durchzuführen und bei Bedarf eine Repeat–Funktion bei längerem Dauerhalten zu starten.

3.3.8.2 Erkennung eines einfachen Klicks (Tastendrucks)

Wie in Aufgabe 3.2 erfahren, kann die „naive“ Auswertung von Tastenereignissen zu Problemen bei der Interpretation führen.

- Probieren Sie aus, wie oft Sie pro Sekunde mit dem Finger eine der Tasten wiederholt drücken können! (Um die Hardware zu schonen, können sie auch auf die Tischplatte klopfen und lassen Sie Ihren Teampartner zählen lassen. Es eignet sich vermutlich ein über mehrere Sekunden bestimmter Mittelwert.
- Definieren Sie nun, welches Aktivitätsmuster über der Zeit (zeichnen Sie ein Diagramm mit Zeitachse) Sie als einen einzelnen Klick interpretieren wollen
- Legen Sie fest, ab welcher „Dauer“ Sie ein langes Drücken zwecks Einschaltens einer Repeat–Funktion erkennen wollen.
- Legen Sie fest, bei welchem Verlauf des Drückens Sie einen Doppelklick erkennen wollen.
- Legen Sie fest, wie sie mit dem „Prellen“ des Tasters zu Beginn oder zum Ende des Drückens umgehen wollen.

Definieren Sie nun einen Datentyp, der für einen Tastendruck steht und Informationen zur gedrückten Taste und der Art des Drückens enthält.

Überlegen Sie zusätzlich, mit welchen LEDs Sie die erkannten und bewerteten Eingabeereignisse im späteren Test darstellen wollen. Legen Sie diese Testprozedur in Ihrem Praktikumsbericht nieder!

3.3.8.3 Erkennung eines Doppelklicks

Nutzen Sie die Vereinbarungen aus Abschnitt 3.3.8.2, um die Erkennung des Doppelklicks zu implementieren. Nach einem Doppelklick soll die LED 4–mal schnell Blinken.

3.3.8.4 Erkennung eines „langen Drückens“

Implementieren Sie abschließend mit den Vereinbarungen aus Abschnitt 3.3.8.2 das „lange Drücken“ und stellen Sie durch Auswertung des Timers ein Repeat–Intervall von 1s bereit. Sie können hier z.B. ein langsames Blinken der Leuchtdioden nutzen.

HINWEIS: stellen Sie sicher, dass alle Ihre Ergebnisse sich durch Tests bestätigen lassen. Gehen Sie in Ihrem Praktikumsbericht auf die Funktionen, aber auch auf die Limitationen Ihrer Implementierung ein. Die folgenden Fragen stellen nur Beispiele dar.

- Funktionieren alle Tasten unabhängig voneinander?
- Kann es vorkommen, dass die Bearbeitung einer Taste eine andere blockiert?
- Was passiert, wenn im Repeat–Modus zusätzlich eine weitere Taste gedrückt wird?
- Was passiert, wenn im Rhythmus des Doppelklicks zwei unterschiedliche Tasten nacheinander gedrückt werden?
- Wie behandelt Ihre Software einen dreifach Klick?

3.4 Konfiguration der seriellen Schnittstelle mit Terminal

3.4.1 Allgemeines zur Funktion der seriellen Schnittstelle

Serielle Schnittstellen übertragen Daten bitweise bei einer vordefinierten Datenrate. Es werden asynchrone und synchrone Schnittstellen unterschieden. Bei der als asynchrone bezeichneten Variante wird kein Taktsignal zwischen Sender und Empfänger übertragen, bei einer synchronen Variante schon.

Zur Kommunikation mit einem Terminal, dem über die USB–Schnittstelle verbundenen PC, wird in unserem Fall eine asynchrone Schnittstelle verwendet. Im Kontext von Microsoft Windows werden diese Schnittstellen in der Regel als „COMx“ bezeichnet, wobei „x“ der laufende Index der verwendeten Schnittstelle ist.

Für die erfolgreiche Kommunikation über die asynchrone Schnittstelle ist es notwendig, dass die beiden Seiten sich vor dem Beginn der Übertragung auf eine gemeinsame Übertragungsgeschwindigkeit einigen. Zusätzlich ist das passende Datenformat auszuwählen. In unserem Fall gibt es hier nur die Möglichkeit der so genannten Start–Stopp Übertragung, die in Abbildung 5**Abbildung 4** dargestellt ist.

Start	B0	B1	B2	B3	B4	B5	B6	B7	P	Stopp

Abbildung 5: Format der Start–Stopp Übertragung

Die Bits B0 bis B4 in Abbildung 5 stellen die eigentlichen Datenbits da. Die Übertragung erfolgt von links nach rechts, also mit dem Least–Significant–Bit (LSB) zuerst.

Zu Beginn der Übertragung wird ein so genanntes Start Bit übertragen, dass immer den binären Wert „0“ hat. Die Übertragung eines Zeichens aus 8 Bit wird mit einem Stopp–Bit abgeschlossen, das immer den Wert „1“ trägt. Damit kann, nach Abschluss eines Zeichens, der Beginn der nächsten Übertragung immer am 0–1 Übergang auf der Zeichenleitung erkannt werden.

Das „P“–Bit i ist optional. Es kann hier ein Paritätsbit zu Sicherung übertragen werden. Wird diese Option genutzt, so besteht die Wahl zwischen „odd“ und „even“ parity. Im ersten Fall (even, gerade) wird P so gewählt, dass die Anzahl der „1“–Bits zwischen Start und Stopp gerade ist, im anderen Fall ungerade.

Sie werden bei der Konfiguration der Schnittstelle diese Auswahlmöglichkeiten in den Konfigurationsregistern finden.

3.4.2 Einstellung und Prüfung der Schnittstelle

Finden Sie zunächst an Hand der Schaltpläne heraus, welche der verfügbaren Schnittstellen für die Kommunikation mit dem PC via des USB–Wandlers genutzt wird.

Stellen Sie diese Schnittstelle nun so ein, dass sie mit 19200 Bit/s, 8–Bit Zeichenlänge und ohne Paritätsbit betrieben wird.

Nutzen Sie nun die Timer–Funktionen (ohne die bisherigen Funktionalitäten zu verändern), um pro Sekunde einen Buchstaben Ihrer Wahl an den PC zu senden.

Stellen Sie jetzt auf dem PC mit der Software „hterm“ eine Verbindung in identischer Konfiguration her. Sie sollten nun im Sekundenrhythmus den gewählten Buchstaben angezeigt bekommen.

3.4.3 Ausgabe verschiedener Zeichen

Die Zeichenausgabe zum PC soll nun in mehreren Schritten leistungsfähiger werden. Damit kann dann in weiteren Arbeitsaufgaben eine Debug–Funktion implementiert werden, um z.B. das Erreichen bestimmter Ablaufpunkte in der Software zu dokumentieren.

a) Verändern Sie die Ausgabe so, dass Sie als Reaktion auf ein Tastenereignis ein bestimmtes Zeichen zum PC senden. Senden Sie unterschiedliche Zeichen auch für Doppelklicks. Wählen sie z.B. Kleinschreibung für einen einfachen Klick, Großschreibung für einen Doppelklick.
b) Sorgen Sie dafür, dass bei langem Drücken mit der Repeat–Funktion Zeichen rhythmisch ausgegeben werden.

Hinweise zur Implementierung finden Sie im Kapitel 15.2. des Family User Guides.

3.4.4 Ausgabe von Zeichenketten

Die Einzelzeichen sollen nun durch ganze, zusammenhängende Textketten ersetzt werden. In der Wahl Ihrer Text sind Sie grundsätzlich frei, jedoch sollten die Text mindestens 4 Zeichen lang sein und immer mit der Zeichenfolge „\x0D\x0A“ (Zeilenumbruch) enden.

a) Geben Sie Texte entsprechend der Funktionen aus Aufgabe 3.4.3 aus.

Vermutlich werden Sie schnell auf Schwierigkeiten stoßen, Ihre Textausgabe zu organisieren und evtl. auch den Verlust von Zeichen oder den Empfang ungültiger Zeichen sehen.

Um diese Probleme zu lösen, gehen Sie schrittweise wie folgt vor:

a) Definieren Sie einen Puffer, z.B „unsigned char Buffer[256]“, in den Sie bei einem Sendewunsch zunächst die gesamte Zeichenkette speichern.
b) Definieren Sie zwei Variablen „int readpos, writepos;“ Diese sollen als Merker dienen, wo die nächsten Daten geschrieben werden dürfen (writepos) und ab wo der eigentliche „Sender“ noch lesen muss (readpos).

Die Funktion des Puffers muss dann wie folgt implementiert werden:

- Mit jedem in den Puffer geschriebenen Einzelzeichen muss „writepos“ um eine Position erhöht werden.
- Erreicht „writepos“ einen Wert größer oder gleich der Länge des Puffers, so wird er auf „0“ zurückgesetzt.
- Das nächste zu sendende Zeichen im Puffer wird durch „readpos“ indiziert. Nach dem Sendevorgang wird readpos inkrementiert
- Erreicht „readpos“ einen Wert größer oder gleich der Länge des Puffers, so wird er auf „0“ zurückgesetzt.
- Gilt „readpos == writepos“, so ist der Puffer leer und das Senden von Zeichen ist zu stoppen.

ACHTUNG: Ein Schreibvorgang muss abgebrochen werden, wenn nach Beendigung die „Leer“-Bedingung einträte. Das der Puffer immer endlich ist und Schreib- sowie Lesevorgang keinen festen Bezug zueinander haben, ist ein Überlastfall grundsätzlich möglich und muss unbedingt abgefangen werden.

Implementieren Sie nun noch folgende Funktionen:

c) Eine Funktion „int send(unsigned char*, int)“, mit der eine Zeichenkette in den Puffer geschrieben werden soll. Diese Funktion soll folgende Werte zurückgeben:

„n“, wenn das Eintragen von „n“ Zeichen erfolgreich war.

Wenn beim Aufruf der Funktion der Puffer noch leer war, so ist der eigentliche Sendevorgang zu starten, d.h. der Interrupt der seriellen Schnittstelle zu aktivieren.

d) Einen Interrupt–Handler für die Bedingung „TxBufferEmpty“. Immer, wenn die serielle Schnittstelle mit diesem Interrupt meldet, dass Sie ein neues Zeichen aufnehmen kann, soll das nächste Zeichen in den Sendepuffer geschrieben werden. Ist der Sendepuffer nach dem Schreibvorgang leer, dann soll der Interrupt abgeschaltet werden.

3.4.5 Empfang von Zeichenketten

Nun sollen auch Zeichen vom PC empfangen werden können.

Implementieren Sie in entsprechender Form einen Empfangspuffer, der ausschließlich durch die Behandlung des Empfangsinterrupts gefüllt wird. Wird die Zeichenfolge „\x0d\x0a“ empfangen[2], so soll im Interrupthandler ein Flag (eine globale Variable) gesetzt werden. Diese globale Variable ist in der Hauptschleife Ihres Programms abzufragen und, sofern gesetzt, eine passende Bearbeitungsroutine aufzurufen.

Die Bearbeitungsroutine sollte entsprechend reagieren. Geben Sie z.B. beim Drücken der Taste 1 „T1pressed“ zum PC aus (zusätzlich zur bereits implementierten LED Aktion), so sollte der Empfang dieses Textes eben dieselbe LED Aktion auslösen.

- Bitte beachten Sie, dass hier auch immer Fehler auftreten können. Stellen Sie sicher, dass unlesbare Texte verworfen werden und (z.B. durch einen Timer) nach einer vorgegebenen Zeit unvollständige Texte aus dem Puffer entfernt werden.
- Senden Sie im Falle einer erfolgreichen Bearbeitung „OK“ an den PC zurück. Tritt ein Fehler bei der Interpretation oder ein Timeout auf, so senden Sie bitte eine entsprechende Reaktion zum PC Terminal.

[2] Die Notation „\x0d\x0a“ fügt zwei hexadezimal angegebene Bytes in den String ein. Hier handelt es sich um die Äquivalente zu „Zeilenvorschub“ (LF, \n, \x0d) und „Wagenrücklauf“ (CR, \r, \x0d)

3.5 Serielle Programmierung der LED–Anzeige

3.5.1 Aufbau und Ansteuerung des LED–Balkens

Der LED Balken ist als Kette von Leuchtdioden des Typs „WS2812b“ ausgeführt. Lesen Sie für den Einstieg in die Programmierung das im Moodle verfügbare Datenblatt.

HINWEIS: Bei genauerem Hinsehen stellt man fest, dass die erwartete Sequenz des WS2812b sich aus 3 Schritten zusammensetzt. Jeder Wert beginnt mit einer Phase von ca. 40µs auf „1“. Für die Übertragung einer „1“ folgen dann zwei „0“ Schritte gleicher Länge, für die Übertragung einer „0“ noch die Folge „10“.

Für die Ansteuerung der LED–Kette ist die serielle Schnittstelle UCA0 im SPI Modus vorgesehen. (Signal BB_Data). Ergänzen Sie Ihre Hardware–Initialisierungsfunktion um die Konfiguration dieser Schnittstelle und der zugehörigen Ports für den Betrieb als SPI Master mit 3 Leitungen! Beachten Sie dabei auch die Konfiguration der Übertragungsrichtung (MSB oder LSB–first) und legen Sie diese geeignet fest.

3.5.1.1 Bestimmung der Grundeinstellungen

Bestimmen Sie die Datenrate, die einer Bit–Dauer von ca. 0.4–0.5µs entspricht. Aus welchem Takt und mit welchem Vorteiler im Baudratengenerator sie eben diese Datenrate für eine serielle Schnittstelle einstellen können. Stellen Sie die Schnittstelle auf diese Geschwindigkeit ein. Beachten Sie dabei auch im Schaltplan „Upperboard“ und „Basisboard“, wie das Steuerungssignal der intelligenten Leuchtdiode zugeführt wird und leiten Sie daraus ab, wie der Datenstrom für eine erfolgreiche Funktion codiert werden muss.

3.5.1.2 Codierung der LED–Beleuchtung

Programmieren Sie eine Funktion, mit der Sie die einzelnen LEDs des Balkens als Array von 10 RGB–Werten ansprechen können. Verwenden Sie dafür den Datentyp und ein entsprechendes Datenfeld wie folgt:[3]

```
typedef struct __TColor  {
     unsigned char green;
     unsigned char red;
     unsigned char blue;
} TColor;
```

Weitere Hinweise finden Sie unter dem Bereich „Code-Schnipsel“ im Moodle

Vor dem Versand müssen Sie nun eine Codierung der einzelnen Werte vornehmen. Beachten Sie, dass Sie pro 8 Bit jeweils 24 Schritte in der oben vorgestellten Codierung versenden müssen.

---

[3] Im Moodle Arbeitsraum finden Sie eine Datei „rgb_led.h“ als Vorlage für eine mögliche Beschreibung der LED-Komponente. Beachten Sie die bedingten Codierungen mittels #ifdef Anweisungen. In Ihrem Projekt werden keine der Symbole definiert sein. Ermitteln Sie also aus den Schalplänen, ob sie diese in Ihrem Fall benötigen werden oder nicht.

Schreiben Sie eine Methode, die jedes Byte in das entsprechende 3–Byte Codierwort überführt. Rufen Sie diese Methode im Sender–Interrupt (UCTXBUF–empty) immer dann auf, wenn Sie ein neues Element übertragen wollen.

HINWEIS: Für die Übertragung aus dem Interrupt heraus ist es sinnvoll, die zu übertragenden Daten durch einen Pointer zu referenzieren, der in der main–routine zusammen mit einer Längenangabe gesetzt wird. Nach dieser Aktion soll dann der Sender–Interrupt freigegeben werden. Der Rest der Übertragung wird dann ausschließlich über den Sender–Interrupt abgewickelt.

HINWEIS: Am Ende der Übertragung soll der Interrupt den Eintrag der zu sendenden Länge löschen, d.h. auf „0“ zurücksetzen und sich selbst abschalten. Durch eine Abfrage der Form „if (Sendelänge == 0)“ kann so in der zentralen Schleife festgestellt werden, ob eine weitere Einstellung des LED Balkens möglich ist oder ob weitere Einträge noch gesperrt sind.

3.5.1.3 Aktivierung der LED Reihe

Ist die Codierung Ihrer Datentypen und Funktionen abgeschlossen, soll nun versucht werden, die Leuchtdioden tatsächlich zu aktivieren. Bitte beachten Sie weiterhin, dass vor der Ansteuerung der LED-Reihe diese nach dem Einschalten oder einem Systemneustart in einen definierten Zustand gebracht werden muss. (Siehe auch „Reset“ in der LED Beschreibung). Nach jeder Ansteuerungsoperation muss ein solcher Reset erneut erfolgen, um die nächste geplante Einstellung auch laden zu können.

HINWEIS: Für die Nutzung der LED-Reihe ist eine zusätzliche Spannungsversorgung über das beiliegende Steckernetzteil herzustellen!

HINWEIS: Es kann, abhängig von der Struktur Ihrer erarbeiteten Software, zu unerwarteten Ausgaben kommen. Sollte dies auch nach Ausschluss aller anderen Fehlerquellen der Fall sein, so ziehen Sie ggfs. den Aufgabenpunkt. 3.5.4, Nutzung der DMA–Funktion zur Entlastung des Prozessors, vor.

HINWEIS: Es wird nicht möglich sein, die Daten Byte-für-Byte zu codieren und dann zu versenden.

Schätzen Sie die Zeit ab, die Sie zur Codierung eines Datenbytes benötigen und vergleichen Sie diese mit der Zeit, die zwischen dem Versand zweier Bytes über die SPI Schnittstelle maximal vergehen darf! Die Übertragung auf der BB-Data Leitung darf innerhalb einer Sequenz nie unterbrochen werden!

3.5.2 Programmierung eines Lauflichts

Durch die interrupt–getriebene Abwicklung des Sendens an die LEDs ist die Hauptschleife des Programms nun stark entlastet und steht für neue Aufgaben zur Verfügung:

- Erzeugen Sie eine Kopie der LED–Daten, auf der während des Sendevorgangs gearbeitet werden kann.

- Modifizieren Sie diese Kopie nach dem Start des Sendevorgangs so, dass sich insgesamt die Funktion eines rotierenden Lauflichts ergibt. Dabei sollen immer nur 4 LEDs gleichzeitig leuchten und sich diese Vierergruppe bewegen.
- Nutzen Sie einen Timer, um einmal pro Sekunde eine Übertragung an den LED–Balken zu starten.
- Nutzen Sie die Tasten „0“und „1“ wie folgt:
  - Taste 0: Umschalten des Leuchtbalkens an/aus.
  - Taste 1: Wechsel der Richtung des Lauflichts, wenn der Leuchtbalken an ist.

3.5.3 Veränderung der Farbe über Tastendruck

Zur weiteren Konfiguration des Lauflichts ändern Sie die Eingabe wie folgt:

- Taste 0, Einfachklick, Umschalten an/aus
- Taste 0, Doppelklick: Umschalten der Richtung
- Tasten 1–3 als Eingabe R / G / B. dabei jeweils
  - Einfachklick: Erhöhung des jeweiligen Wertes um 1
  - Doppelklick: Erhöhung des jeweiligen Wertes um 25 oder auf max. 255
  - Langes Drücken: Erhöhung des Wertes alle 100ms

3.5.4 Nutzung der DMA–Funktion zur Entlastung des Prozessors

Zum Abschluss dieser Aufgabe soll die Arbeit der eigentlichen CPU weiter reduziert werden. Dazu soll die Funktion des „Direct–Memory–Access“ (DMA) genutzt werden. Lesen Sie dazu im Kapitel 6 des Family User Guides nach, wie die DMA Einheit konfiguriert werden muss, um die Daten in das Senderegister der UCA0 Schnittstelle ohne weiter CPU–Interaktion eintragen zu können.

4 Weiterführende Aufgabenstellungen

4.1 Initialisierung des LCD Displays

4.1.1 Arbeitsweise des LCD Displays

Die Funktion des LCD Displays ist im Dokument „Datenblatt LCD Controller“ beschrieben. Entsprechend der in den Schaltplänen gezeigten Anordnung wird das Display über einen 8 Bit breiten Datenbus angesteuert, der direkt an Port 4 des Controllers angeschlossen ist. Darüber hinaus werden 3 Steuerleitungen direkt vom Controller aus angesprochen. Dies sind:

- Die Auswahlleitung #E (Port 6, Bit 3)
- Die Steuerleitung für die Richtung des Transfers (Port 2, Bit6)
- Eine Adressleitung zur Selektion von Registern oder Daten (Port 6, Bit 4)

4.1.2 Konfiguration des LCD−Displays

Implementieren Sie die Initialisierung des LCD Displays entsprechend des Ablaufes des Dokuments „Datenblatt des LCD Controllers“. Nutzen Sie für die Festlegung von Verzögerungen (Delays) den bereits zuvor implementieren, zyklischen Timer. Beachten Sie dabei:

- Die angegebenen Zeiten sind Mindestzeiten.
- Bei der Aktivierung der Steuerleitungen sind immer nur einzelne Bits in den jeweiligen Ports betroffen. Nutzen Sie daher die Zuweisungen „&=“ bzw. „|=“.
- Wenn Sie Daten vom Display Controller lesen wollen, so nutzen Sie für die Umschaltung immer die folgende Reihenfolge:
  i. Umschalten des Port 4 auf Input (alle Bits)
  ii. Setzen des R/W Bits auf Lesen (LCD Controller treibt den Bus)
  iii. <Durchführen des Lesevorgangs>
  iv. Setzen des R/W Bits auf Schreiben (LCD Controller nun passiv)
  v. Umschalten des Port 4 auf Ausgang (alle Bits)

4.1.3 Verwendung des LCD−Displays

Implementieren Sie eine Routine, die auf Tastendruck einen beliebigen Buchstaben auf das Display ausgibt. Ein weiterer Tastendruck soll einen Buchstaben hinzufügen. Sind mehr Buchstaben eingegeben worden, als das Display breit ist, so sind die folgenden Zeichen in die zweite Zeile zu schreiben.

Es bietet sich hierbei an, folgende Methoden zu implementieren:

- Eine Methode zum Löschen des Displays
- Eine Methode zum Scrollen des Displays nach links und rechts
- Eine Methode zum Positionieren des Cursors
- Eine Methode zur Ausgabe eines Einzelzeichens mit Rückmeldung, ob der Rand des Displays erreicht wurde.

4.1.4 Ausgabe von Texten auf dem LCD Display

In Ergänzung zu den Ausgabemöglichkeiten aus 4.1.3 soll nun eine Methode codiert werden, mit der Zeichenketten (Strings) auf das Display ausgegeben werden können. Die Signatur einer solchen Funktion soll dem Muster „void LCD_WriteString (const char* text)“ folgen.

4.2 Konfiguration und Nutzung des A/D Wandlers

4.2.1 Allgemeines zum A/D Wandler

MCUs werden häufig zum Messen und Regeln verwendet. Messwerte (wie z.B. Temperatur, Helligkeit, Lautstärke etc.) liegen dabei in der Regel zunächst als analoge Spannungen vor, die durch einen entsprechenden Sensor bereitgestellt werden. Die Aufgabe des A/D–Wandlers ist es nun, diese analoge Größe in einen ganzzahligen Zahlenwert umzuwandeln.

Die MCUs der MSP403Fxxx Serie bieten dazu einen Wandler an, der einen gegebenen Spannungswert in eine 12 Bit lange Zahl wandelt. Hierbei werden Referenzspannungen für den Wert „0“ und den Wert „4095“ eingestellt und der Messwert auf einer linearen Skala zwischen diesen beiden Werten bestimmt.

Dabei kann zwischen max. 16 Eingängen (siehe Kapitel 23 des MSP403 Family User Guide) ausgewählt werden, von denen aber nur 8 an externe Pins der MCU herausgeführt sind.

- Photo–Diode: A0
- Temperatursensor: A1
- Mikrofon: A2
- Infrarotdiode: A5

4.2.2 Das Setup des A/D Wandlers

Beim Setup des A/D Wandlers sind die Anweisungen des Kapitels 23 zu befolgen. Gehen Sie zunächst so vor, dass Sie eine einzelne Messung starten und diese durch einen Tastendruck auslösen. Verwenden Sie als Quelle zunächst den internen Temperatursensor der CPU. Implementieren Sie eine Methode, die den gemessenen Spannungswert in einen Zielwert konvertiert. Geben Sie den berechneten Wert als Text auf das LCD Display im Format „##.# °C“ aus, wobei ‚#‘ jeweils für eine dezimale Ziffer steht.

4.2.3 Konfiguration des externen Temperatursensors

Entsprechend dem Vorgehen aus 4.2.2 wird nun der externe Temperatursensor ausgewertet. Entnehmen Sie die Spannungs–Temperaturkennlinie aus dem Datenblatt.

4.2.4 Nutzung des Helligkeitssensors

Gehen Sie entsprechend 4.2.4 vor und ermitteln Sie Energieeinstrahlung auf die Photodiode. Die ausgegebene Helligkeit soll mit der Einheit W/A angezeigt werden.

4.2.5 Nutzung des Mikrofons als Geräuschmelder

Das Mikrofon detektiert Töne im Bereich von 50–5000Hz. Daher ist der Messwert in der Regel stark schwankend. Experimentieren Sie mit unterschiedlichen Sampling–Intervallen und Mittelwertbildung über mehrere Messwerte und notieren Sie die Ergebnisse. Leiten Sie daraus eine Beschreibung über die Charakteristik Ihrer Geräuschmessung ab. Beantworten Sie sich selbst dabei Fragen wie:

i) Welche Arten von Geräuschen sollen erkannt werden?
ii) Welche Empfindlichkeit soll die Messung haben.

iii) Wie oft kann/soll die Messung wiederholt werden.

Das Ergebnis dieser Aufgabe ist offen, jedoch sollte ein Test nach Ihren Parametern durch den Versuchsleiter zum Erfolg führen!

4.3 Ausgabe der Systemaktivtäten auf ein Terminal

4.3.1 Allgemeines zur Aufgabenstellung

In dieser Teilaufgabe sollen Sie die Messwerte wie Temperatur, Helligkeit, und Geräuscherkennung auf ein Terminal ausgeben. Jeder Messwert soll regelmäßig wiederholt werden.

Nutzen Sie dazu den Timer B für das Auslösen des Messvorgangs. Die Temperatur soll dabei jede Sekunde gemessen werden, die Helligkeit und der Geräuschpegel alle 50ms. Die Ausgabe der letzten beiden Werte soll wahlweise als Mittelwert oder als Spitzenwert erfolgen. Schalten Sie durch das Drücken einer beliebigen Taste zwischen den Darstellungen um ändern Sie den ausgegebenen Text für die jeweiligen Sensoren.

Die Texte zur Darstellung auf dem Terminal sind grundsätzlich frei wählbar.

4.4 Ansteuerung von Aktuatoren mittels eines PWM–Signals

4.4.1 Allgemeines zur PWM

Das Akronym PWM bedeutet „Pulsweiten–Modulation“. Darunter ist zu verstehen, dass eine bestimmte Stellgröße ´mit der Breite (Zeitdauer) einer in regelmäßigen Abständen wiederkehrenden Impulses assoziiert wird.

Für elektrotechnisch Anwendungen, wie z.B. die Steuerung der Helligkeit einer Beleuchtung, ist dies gleichbedeuten mit der Veränderung des Effektivwertes einer periodischen Funktion.

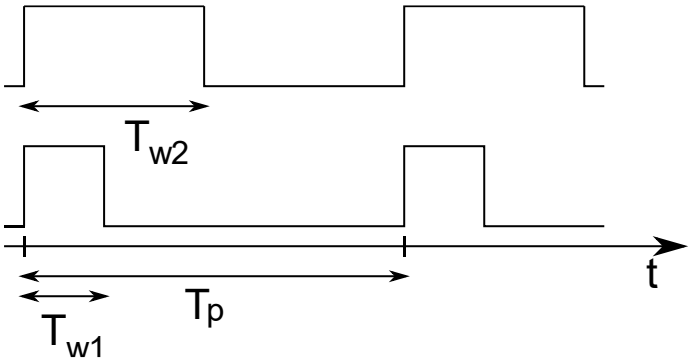

Abbildung 6: Prinzip der Pulsweiten–Modulation

Die Abbildung 6 zeigt hierbei ein solches Beispiel. O.B.d.A können wir in der Vertikalen Richtung das Signal, also z.B. eine Spannung oder Stromstärke, annehmen. Über die Periode hinweg gibt es also eine „an“ und eine „aus“ Phase. Das Verhältnis dieser Phasen bzw. die Länge der „an“-Phase entspricht dann der zu übertragenden Information.

Die Timer des MSP430 stellen explizit Funktionen zur Erzeugung solcher PWM Signale zur Verfügung, die in den kommenden Aufgaben genutzt werden sollen.

4.4.2 Ansteuerung eines einfachen Lüfters

In diesem geht es grundsätzlich um die Einstellung einer Effektivleistung. In Anlehnung an GET–A erscheint es plausibel, dass ein Elektromotor mit steigender Leistung immer schneller dreht. Die maximale Leistung steht offensichtlich zur Verfügung, wenn die „an“-Phase auf Dauer anliegt.

HINWEIS: In Ihrer Versuchsbox finden Sie 3 Lüfter. Einen mit 2-poligem Anschluss, einen Lüfter mit USB-Stecker und einen dritten mit blauem Stecker, der in einem eigenen Halter montiert ist. Nur die ersten beiden sind im Versuch per PWM anzusteuern!

Die Lüfter sollen am Ende in ihrer Drehzahl steuerbar sein und den Eindruck eines gleichmäßigen Laufs vermitteln. Verwenden Sie zur Einstellung einen Timer und die passende Funktionszuordnung an den Ausgangspins zum Motortreiber.

Gehen Sie bei diesem Experiment schrittweise vor und versuchen Sie, die folgenden Fragen für die Lüfter 1 und 2 zu beantworten:

a) Wie lange läuft der Lüfter nach, wenn Sie die Versorgungsspannung abschalten?

b) Welche Schlussfolgerungen ziehen Sie daraus für die Periodendauer Ihres PWM Signals?
c) Bei welcher Periodendauer t_p erhalten sie für ein Verhältnis an/aus = 1:1 einen gleichmäßigen Laufeindruck?
d) Ab welchem minimalen Verhältnis an/aus läuft der Lüfter gar nicht mehr an?
e) Verändert sich das Verhältnis aus d), wenn Sie die Periodendauer verändern?
f) Erläutern Sie die beobachteten Effekte aus Ihren Kenntnissen um Elektrotechnik und Mechanik. (Insbesondere Induktion und Massenträgheit).

4.4.3 Wechsel der Lüfterdrehzahl auf Tastendruck

In Ergänzung zum Aufgabenteil 4.4.2 sollten Sie die Lüfterdrehzahl durch die Tasten auf der Experimentierbox verändern können. Alternativ können Sie diese Funktion auch durch das Versenden von Kommandos vom PC aus implementieren.

Ihre Aufgabe besteht darin, ein kleines „User-Manual“ zu beschreiben, dass den Bedienvorgang mit den zu erzielenden Effekten eindeutig beschreibt und dann eben diesen Bedienvorgang auch zu implementieren.

4.4.4 Ansteuerung des Lautsprechers

In diesem Aufgabenteil soll die PWM Ausgabe auf den Lautsprecher der Experimentierbox geleitet werden. Offensichtlich werden sich so Töne ergeben, bei denen die Periodendauer die Frequenz des Tons bestimmt.

Was verändert sich, wenn Sie neben der Frequenz auch das Tastverhältnis modifizieren?

Können Sie die Lautstärke verändern? Wenn ja, wie?

4.4.5 Mini–Piano

Das Sie nun gezielte Frequenzen ausgeben können, sollen nun 4 benachbarte Töne aus der C-Dur Tonleiter auf die Tasten gelegt werden. Wenn Sie kein Instrument spielen, lassen sich diese Frequenzen leicht im Internet finden.

Sind Sie etwas experimentierfreudiger, gelingt es Ihnen vielleicht, eine einfache Melodie wie in einer Spieluhr ablaufen zu lassen. Die entsprechenden Parameter einzelner Töne einzustellen haben Sie sich ja im vorigen Teil erarbeitet.

4.4.6 Ansteuerung eines Servomotors

ACHTUNG: Vermeiden Sie eine Verpolung des Sevormotors! Kontrollieren Sie vor dem Betrieb noch einmal, ob mit der gewünschten Polarität der USB-Lüfter dreht. Falls ja, ist die Polarität auch am Servo-Anschluss korrekt.

Servomotoren finden in unterschiedlichen Anwendungsbereichen Anwendung. Allgemein wird dabei der Begriff des Servomotors häufig dort verwendet, wenn der Motor eine Stellfunktion bedient. In unserem Fall ist dies keine Rotation mit einer regelbaren Drehzahl, sondern die Einstellung eines Drehwinkels rechts und links einer Nullstellung.

Der in Ihrer Experimentierbox vorliegende Servomotor stammt aus dem Modellbau und wird in dieser Form in fernsteuerbaren Modellen eingesetzt. Die Einstellung erfolgt über ein PWM Signal.

Suchen Sie im Internet nach dem Parameter für t_p und den Limits für t_w! Bevor Sie den Servomotor anschließen, kontrollieren Sie bitte mit Hilfe eines Oszilloskops, ob die gewünschte Signalform auch tatsächlich erzeugt wird, da grobe Abweichungen zur Zerstörung des Motors führen können.

Stellen Sie nun, ähnlich wie in 4.4.3, den Drehwinkel auf Knopfdruck auf vorgegebene Werte ein!

Wo wir gerade bei Musik waren: Nutzen Sie den Servo doch mal als Metronom!

4.4.7 Betrieb eines Lüfters mit Tacho

Der Lüfter mit dem blauen Stecker bietet am weißen Kabel ein Tachosignal an. Dieses erwartet, dass der Microcontroller einen Eingang mit Pullup-Widerstand anbietet. Ziel ist es, die Frequenz dieses Signals auszuwerten.

Da das Tachosignal von der Versorgungsspannung abhängt, ist eine Pulsweitenansteuerung in einem solchen Fall nicht sinnvoll. Daher wird die Drehzahl des Lüfters über eine Variation der Versorgungsspannung ermöglicht.

4.4.7.1 Bestimmung des Drehzahlsignals

Im Leerlauf (siehe Datenblatt) sollte die Drehzahl bei einer Spannung von ca. 5V (Voreinstellung) bei ca. 8000min^{-1} liegen. (Siehe Datenblatt). Ermitteln Sie aus dieser Angabe und Ihrer Messung, wie viele Impulse pro Umdrehung das Tachometer liefert. Nutzen Sie dazu einen Timer, um die Periodendauer des Tachosignals zu bestimmen.

4.4.7.2 Variation der Versogungsspannung

Der in den Schaltplänen des Breakout-Boards dargestellte Schaltspannungsregler misst über einen Spannungsteiler vor der Feedbackleitung (FB) die erzeugte Ausgangsspannung und regelt dann (siehe Datenblatt) auf ca. 1.2V am Feedback-Eingang nach. Durch Modifikation des Teilerverhältnisses per Digitalpotentiomenter kann die Spannung nun zwischen ca. 5V und 2.5V verändert werden. Sie finden die Datenblätter z.B. bei **www.mouser.de** unter den Typkennzeichen der Bauteile.

1. Benutzen sie die I2C Funktion der seriellen Schnittstelle, um mit dem Digitalpotentiometer zu kommunizieren.
2. Stellen Sie zunächst willkürlich Widerstandswerte ein und beobachten Sie den Effekt am Lüfter. (Sie können hier auf Grund der Parallelschaltung keinen linearen Verlauf erwarten)
3. Erstellen Sie zu Abschluss eine Kennlinie Drehzahl als Funktion des eingestellten Widerstandswertes.

4.5 Aufbau eines Funkuhrempfängers mit dem DCF77 Modul

4.5.1 Beschreibung der DCF77–Funktion

Das DCF–77 Signal ist ein Signal, dass in der Nähe von Frankfurt (bei Seligenstadt) auf der Frequenz 77.5kHz abgestrahlt wird. Es handelt sich hierbei um eine Pulse–Code–Modulation, bei der die Informationen über eine zeitweise Verringerung des Signalpegels übertragen wird. Grundsätzlich gilt hierbei:

- Binäre „1“ wird als Absenkung für 200ms übertragen
- Binäre „0“ wird als Absenkung für 100ms übertragen
- In den Sekunden 0–58 jeder Minute wird ein Bit übertragen
- In Sekunde 59 findet KEINE Absenkung statt.[4]

Die Codierung der Bits finden Sie in Tabelle 1. Beachten Sie, dass die Zeitinformationen alle BCD (Binary coded Decimal) übertragen werden. Die xParity Bits sind als gerade Parität berechnet, d.h. die Summe aller „1“ Bits des gesicherten Blocks incl. des Paritätsbits muss gerade sein, um als fehlerfrei erkannt zu werden.

Folgende Sonderfunktionen werden codiert:

- SZCHNG: ist dieses Bit gesetzt, so wird am Ende der aktuellen Stunde von Sommer auf Winterzeit oder umgekehrt umgestellt.
- MESZ: Mitteleuropäische Sommerzeit aktiv
- MEZ: Mitteleuropäische Zeit aktiv
- SSEK: Am Ende der aktuellen Stunde wird eine Schaltsekunde eingefügt.

Im Falle einer Schaltsekunde wird in Bit 59 eine zusätzliche „0“ übertragen. Die darauffolgende Sekunde zeigt dann keine Pegelabsenkung und synchronisiert somit den Start der folgenden Minute.

Die kodierten Zeit– und Datumswerte gelten für die jeweils folgende Minute, d.h. nach einer erfolgreichen Dekodierung dürfen diese mit dem nächsten Minutenstart angezeigt werden.

[4] Offensichtlich handelt es sich hier auch um ein Pulsweiten-moduliertes Signal. Im allgemeinen Sprachgebrauch verwendet man den Begriff PWM immer dann, wenn ausschließlich direkte, physikalische Effekte intendiert sind. Der Begriff PCM hingegen findet sich immer dann, wenn das PWM Signal bei einem Empfänger digital interpretiert wird.

Bit		Bit		Bit	
0	„0“ Start	20	„1“	40	D10
1	X	21	M1	41	D20
2	X	22	M2	42	W1
3	X	23	M4	43	W2
4	X	24	M8	44	W4
5	X	25	M10	45	M1
6	X	26	M20	46	M2
7	X	27	M40	47	M4
8	X	28	MParity	48	M8
9	X	29	S1	49	M10
10	X	30	S2	50	Y1
11	X	31	S4	51	Y2
12	X	32	S8	52	Y4
13	X	33	S10	53	Y8
14	X	34	S2	54	Y10
15	X	35	SParity	55	Y20
16	1: SZCHG	36	D1	56	Y40
17	1: MESZ	37	D2	57	Y80
18	1: MEZ	38	D4	58	DatParity
19	1: SSEK	39	D8	59	—

Tabelle 1: Codierung der DCF77 Bits

Für den Anschluss des DCF–77 Moduls gehen Sie wie folgt vor:

- Stecken Sie das schwarze Kabel auf „GND“.
- Stecken Sie das rote Kabel auf „VCC“.
- Stecken Sie das farbige Kabel auf P2.1.
- Programmieren Sie P2.1 als Input mit Pullup–Widerstand.

4.5.2 Programmierung der Biterkennung

Stellen Sie den Eingang P2.1 so ein, dass für jede Flanke des Eingangssignals ein Interrupt ausgelöst wird. Dazu müssen Sie die Polarität der Flanke in jedem Interrupt invertieren!

Stellen Sie einen Timer so ein, dass sich die Zeit von 200ms gut innerhalb des 16Bit Zählerwertes darstellen lässt.

Geben Sie nun die jede Sekunde erkannten Bitwerte entweder auf dem Terminal oder auf dem LCD–Display aus.

4.5.3 Ergänzung der Minuten–Synchronisation

Messen Sie mit einem zweiten Timer die Zeit zwischen dem Empfang von jeweils 2 Bit und erkennen Sie so die Synchronisation der nächsten Minute. Geben Sie auch dieses Ergebnis entweder auf das Terminal oder das LCD aus.

HINWEISE:

Die Empfangsqualität des DCF–77 Moduls kann abhängig von der Orientierung und Platzierung im Raum drastischen Änderungen unterliegen. Es hat sich als sinnvoll herausgestellt, einen Testmodus zu implementieren, der alle 10ms das Eingangssignal bestimmt und dann z.B. für den vollen Pegel ein Zeichen ‚0' und für den abgesenkten Pegel das Zeichen ‚1' an den PC per serieller Schnittstelle sendet.

Bei hinreichender Datenrate des UART erhalten Sie dann auf dem Terminal 100Zeichen pro Sekunde und sollten schnell erkennen können, ob sich ein PWM Signal erkennen lässt. Im Optimalfall sehen Sie dann 90x'0' und 10x'1' (oder eben 80:20 oder, am Ende einer Minute, eine Pause von 180/190 mal ‚0', gefolgt von 10x'1').

Suchen Sie sich einen Aufstellort, an dem diese Zeichenfolge klar erkennbar und nicht zu stark gestört auftritt.

4.5.4 Auswertung der Zeitinformation

In diesem letzten (und programmiertechnisch anspruchsvollsten) Aufgabenteil geht es nun darum, die Zeit- und Datumsinformation auf dem LCD-Display darzustellen. Eine ideale Lösung könnte folgende Eigenschaften haben:

a) Nach dem Einschalten lässt sich erkennen wie weit die Synchronisation auf das DCF-77 Signal bereits fortgeschritten ist. (Dies hilft später auch bei der Fehlersuche, falls sich das System nicht synchronisiert)
b) Sobald der Minutenrhythmus erkannt worden ist, lassen sich die Sekunden anzeigen.
c) Nach dem Empfang einer vollständigen Minute werden Datum und Uhrzeit dargestellt.
d) Auf Knopfdruck kann man das Datum zwischen DD:MM:YY oder einer Wochentagsdarstellung ohne die Jahreszahl wechseln.
e) Ist die Synchronisation einmal geschafft, sollte auch bei einem Ausfall des DCF-Signals die Uhr frei weiterlaufen! (Eine evtl. Drift zwischen dem Zeitnormal des MSP430 und dem DCF-77 Signal lässt sich ja in einer hinreichen langen Synchronphase bestimmen)

Viel Spaß und Erfolg dabei!

Anhang A, Allgemeines zur Programmierung

A.1 Rechnerkonzepte nach „von Neumann“ oder „Havard“

Computer und Mikrocontroller, wie wir sie heute in praktisch allen Lebensbereichen finden, arbeiten in fast allen Fällen nach einer der in der Überschrift genannten Architekturen.

Die drei grundlegenden Komponenten eines solchen Computersystems sind dabei:

1) Ein oder mehrere Speicher, in der Informationen in binärer Form, so genannten Bits, abgelegt werden. Dieser Speicher ist typisch in Einheiten aus 8 zusammenhängenden Bits, so genannten Bytes organisiert. Um grundsätzlich eine Information (ein Byte) in einem Speicher zu identifizieren, wird die so genannte „Adresse“ des gesuchten Bytes im Speicher, angegeben.
2) Die “Arithmetic and Logic Unit” (ALU), die binäre Operationen wie „Addition“ oder „Vergleiche“ ausführen und Ergebnisse liefern kann.
3) Ein Steuerwerk, dass in der Lage ist, aus dem Speicher Informationen für die ALU bereitzustellen und die Ergebnisse der ALU wieder in den Speicher zurückzuschreiben.[5]

Durch Zusammenwirken von ALU und Steuerwerk entsteht nun die so genannte CPU (Central Processing Unit).

Die Ausführung eines Programms wird dadurch bewerkstelligt, dass Informationen an einer Stelle des Speichers gelesen werden. Diese Informationen werden dann (dies ist der definierte Algorithmus) als Anweisung interpretiert. Durch Decodierung dieser Anweisung kann das Steuerwerk notwendige Daten der ALU zuführen (ggf. dabei Informationen aus anderen Speicherstellen abrufen) und am Ende der Anweisung das Ergebnis bei Bedarf wieder speichern.

Für die Ausführung eines Programmes wird hierbei offensichtlich zwischen Anweisungen (dem so genannten „Code“) und „Daten“ unterschieden.

Im Allgemeinen wird unter „Code“ ist dabei gleich eine Folge von Anweisungen zu verstanden, die von der „CPU“ zunächst in linearer Abfolge ausgeführt werden. Die „Daten“ in diesem Kontext bezeichnen Inhalte, die durch die Anweisungen im Code bearbeitet werden.

Fundamental für die Programmierung ist dabei das Verständnis von „Operatoren“ und deren „Operanden“. Im allgemeinen Fall gilt dabei das in Abbildung 7 dargestellte Denkmodell

[5] Der hier verwendete Speicher muss direkten Zugriff auf Informationen an beliebigen Adressen ermöglichen. Massenspeicher, wie z.B. Festplatten, USB Devices oder DVDs gehören zu den sequentiellen Speichern und sind nicht Gegenstand dieser Betrachtungen.

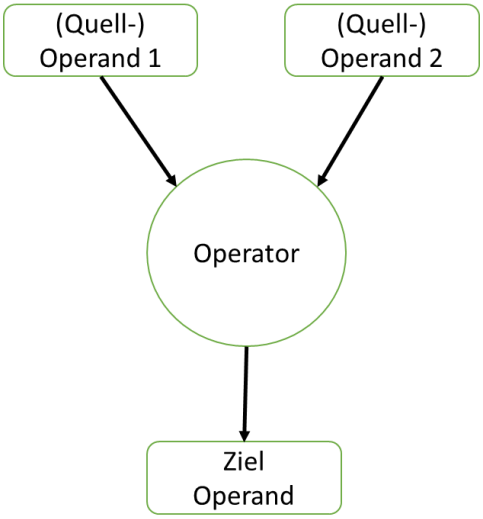

Abbildung 7: Operationen in CPU

Ein Operator verknüpft 2 Quelloperanden (diese werden gelesen) und weist das Ergebnis einem Ziel zu. Letzteres wird dabei „bereitgestellt“.

Beispiel: *Betrachten wir den Ausdruck . Die Quelloperanden sind dabei di Variable „a“ und die Konstante „1“. Wollen Sie diese Gleichung auf Papier lösen, so **lesen** Sie den Wert von „a“ an irgendeiner Stelle auf Ihrem Blatt. Die Konstante „1“ ist direkt innerhalb der Anweisung enthalten. Sie wird ebenfalls gelesen. Die Rechnung geschieht letztlich in Ihrem Bewusstsein. Erst das Notieren des Ergebnisses auf dem Blatt (**die Schreiboperation**) macht das Resultat an einer bestimmten Stelle sichtbar.*

*Mithin ist es offensichtlich so, dass der genannte Ausdruck in zwei unterschiedliche Schritte zerfällt. Zum einen die **numerische Berechnung** des Ergebnisses und zum anderen die **Zuweisung des Ergebnisses*** zu einer Speicherstelle.

Die „Operanden“ in Abbildung 7 stellen die „Daten“ des Programms dar, die Folge der Operatoren „+“ und „=“ die Anweisungsreihenfolge und damit den Code Ihres Programms.

Hinweis: Eine Notation, in der zur Beschreibung einer Operation die Adressen aller Operanden explizit angegeben werden, nennt man auch „3–Adddress–Notation“. Die Konvention ist dabei symbolisch „Operator src1 src2 dst“[6].

In beiden Rechnerarchitekturen wird, entsprechende der Erkenntnisse aus dem Beispiel, zwischen Speicherbereichen für Daten und für Code unterschieden. Diese Unterscheidung ist grundsätzlich nur der Interpretation der Inhalte des verfügbaren Speichers geschuldet.

In der so genannten „Von Neumann Architektur“ ist ein (logisch) gemeinsamer Speicher für Code und Daten vorhanden. Es gibt zu diesem Speicher genau eine

[6] Src1 und src2 stehen dabei für die Quelloperanden 1 und 2. Diese können jeweils direkt in der Anweisung angegeben werden (sog. „Immediate data“) oder die Adresse einer Speicherzelle bezeichnen, deren Inhalt vor der Verarbeitung erst noch gelesen werden muss.

Speicherschnittstelle. Daraus folgt, dass Zugriffe auf den Code und die benötigten Daten nur sequentiell (also nacheinander) erfolgen können.

In der Havard–Architektur werden zwei Speicherschnittstellen bereitgestellt. Dabei ist die eine ausschließlich für das Lesen von Code definiert, während über die andere Daten wahlweise gelesen oder geschrieben werden können. Es erscheint offensichtlich, dass die Havard Architektur leistungsfähiger ist, denn es ist jetzt offensichtlich möglich, berechnete Daten zurückzuschreiben währen ***gleichzeitig*** über die andere Speicherschnittstelle die nächste Anweisung gelesen werden kann.

Bei beiden Architekturen gibt es jedoch keine definierte Aussage, wie die einzelnen Speicherbereiche physikalisch aufgebaut sein müssen. Jedoch wird immer gefordert, das ein willkürlicher Zugriff auf alle verfügbaren Speicherplätze möglich sein muss

In den im Versuch verwendeten Mikrocontrollern ist praktisch der Speicherbereich, in dem Code abgelegt wird, ein nicht flüchtiger Speicher (Flash Memory). Dieser hat folgende Eigenschaften:

- Der Inhalt des Code–Speichers verändert sich auch bei Ausfall der Versorgungsspannung nicht.
- Der Inhalt des Code–Speichers kann während der Programmausführung nicht verändert werden.
- „Programmierung“, d.h. eine Veränderung des Speicherinhalts für Programmcode, findet nur zu Beginn einer Debug–Sitzung in einem einmaligen Programmiervorgang durch das spezifische Programmiergerät statt. Um diesen Vorgang eindeutig zu bezeichnen, wird auch häufig das Wort „Flashen“ verwendet.[7]
- Das Wort „Programmierung“ wird natürlich auch als Bezeichnung für den Vorgang der Erstellung des Quelltextes, also des durch Menschen lesbaren Textes als Beschreibung des gewünschten Programmablaufs verwendet.

Innerhalb der CPU wird nun der Programmcode grundsätzlich linear durchlaufen. Die jeweils nächste, zur Ausführung vorgesehene Anweisung, wird durch den so genannten Programm Counter (PC) indiziert.

Die Begrifflichkeit der „Indizierung“ manifestiert dabei die Auffassung von Speicher als einem universellen Array (hier einer eindimensionalen Matrix) mit Einträgen konstanter Länge. Historisch begründet wird als „Speicherstelle“ immer ein 8–Bit breite Quantität, ein Byte, angenommen. Eine „Adresse“ ist mithin die (fortlaufende) Nummer eines solchen Bytes im Array.

Aus dieser Auffassung lässt sich erkennen, dass zur Ausführung die CPU ein oder mehrere Byte ab der durch den PC vorgegebenen Adresse lädt und dann die an dieser Stelle gespeicherte Anweisung ausführt. Da im Steuerwerk der CPU fest implementiert ist, welche Anzahl von Bytes zu einer bestimmten Anweisung gehören, wird der PC im Laufe der Abarbeitung auf die nächstfolgende Anweisung gesetzt.

[7] Der Begriff des “Flashens” ist vom Namen des modernen, nicht flüchtigen Speichertyps, dem so genannten „Flash–Speicher“ abgeleitet.

A.2 Sprünge, Calls und Vektoren

Da Code in einem linearen Array nur konsekutiv gespeichert werden kann, Algorithmen aber immer auch Entscheidungen „entweder oder“ verlangen, müssen auch Möglichkeiten implementieren, den PC gezielt mit neuen Werten zu versorgen. Damit kann auch eine nicht lineare Ausführung sichergestellt werden.

1. Sprünge
   Unter Sprüngen versteht man die bewusste Veränderung des PC. Der PC wird dabei mit einem neuen Wert geladen. Dieser neue Wert bezeichnet die Adresse, an der die Programmausführung fortgesetzt wird. Es werden dabei unbedingte oder bedingte Sprünge unterschieden. Letztere werden bei manchen CPUs auch als „Branches“ bezeichnet. Im Programm geht einem solchen bedingten Sprung immer die Berechnung einer Bedingung voran. Diese Bedingung wird dann im Sprungbefehl abgefragt und dann, falls die Abfrage als „wahr“ bewertet wird, die im Sprungbefehlt codierte Zieladresse in den PC geladen.
2. Calls (oder Funktionsaufrufe)
   Funktionsaufrufe gleichen den unbedingten Sprüngen. Sie werden immer ausgeführt, wenn sie im Code gelesen werden. Im Unterschied zu einem Sprung (Jump) wird jedoch die Adresse der linear nächsten Anweisung in einem speziellen Bereich des Datenspeichers, dem so genannten Stack (Stapel) gespeichert. Diese „Rücksprungadresse“ liegt nun ganz oben auf dem Stapel. Am Ziel eines Calls muss nun eine Methode/Routine implementiert sein, deren letzte Anweisung ein sogenannter „Return“ Befehl ist. Dieses ist ein spezieller Sprung, der den obersten Eintrag des Stapels in den PC lädt und damit die Ausführung genau an der Stelle NACH dem vorherigen Call fortsetzt.
3. Vektoren
   Der Name „Vektor“ ist allegorisch zu dem Begriff des Vektors aus der Mathematik zu sehen. Vektoren stehen im Speicher und Ihr Inhalt „zeigt“ auf eine andere Speicherstelle.
   Vektoren kommen immer dann zum Einsatz, wenn ein Programm unabhängig von der inneren Logik des Algorithmus an bestimmten Stellen fortgesetzt werden soll. Prominentester Vertreter dieser Art ist vielleicht der Reset–Vektor. Dieser steht an einer (durch den Designer der CPU Familie) vordefinierten Speicherstelle und enthält die Adresse der ersten Anweisung des Anwenderprogramms.
   Weitere Vektoren werden verwendet, um die Ausführung so genannter „Interrupt–Handler“ auszulösen. Interrupts unterbrechen den regulären Programmablauf zu nicht vorhersehbaren Zeiten. Abhängig von der Quelle des Interrupts (die MSP430 CPU kennt bis zu 256 solcher Quellen) wird dann der entsprechende Vektor gelesen und die Ausführung sofort an der im Vektor vermerkten Adresse fortgesetzt.

A.3 Struktur von Assembler–Programmen für den MSC430

Wenn Sie Ihr erstes Assembler–Programm automatisch erzeugen lassen, so finden Sie hier, recht am Anfang der Datei, die folgenden Anweisungen:

```
            .def    RESET                   ; Export program entry-point to
```

```
                                        ; make it known to linker.
;-------------------------------------------------------------------------------
            .text                       ; Assemble into program memory.

RESET       mov.w   #__STACK_END,SP     ; Initialize stackpointer
StopWDT     mov.w   #WDTPW|WDTHOLD,&WDTCTL  ; Stop watchdog timer
```

Die Anweisung „def RESET“ gibt dabei bekannt (deklariert), dass es im Weiteren ein Symbol des Namens „RESET“ geben wird. Nach der Übersetzung des Codes wird der Linker aus diesem Symbol den Reset–Vektor generieren.

Die Anweisung „.text“ gibt an, dass der nun folgende Code in einen Bereich mit dem Namen „text“ einzufügen ist. Arbeiten Sie mit mehr als einer Quelldatei, so muss in jeder Datei vor dem Code diese Anweisung gegeben werden. Auf Grund dieser Benennung kann der Linker beim Zusammenbinden der unterschiedlichen Objekte die Teile des Programms passend zuordnen.

In den folgenden Zeilen finden Sie, in der jeweils **ersten Spalte** der Zeile beginnend, zwei so genannte „Label“, hier „RESET“ und „StopWDT“. In jedem Fall wird die lokale Position im .text Segement mit dem Text des Symbols/Labels assoziiert. In unserem Fall erhielte RESET den Wert „0“, StopWDT den Wert „2“, da die erste Anweisung 2 Byte im Maschinencode belegt. Man sagt, diese Symbole werden hier **definiert**.

Hinweis: Eine Deklaration eines Symbols (eine Bekanntmachung seiner Existenz) darf beliebig oft wiederholt werden. Eine „Definition“ hingegen darf immer nur einmal erfolgen!

Hinweis: Wollen Sie eine Anweisung codieren, so darf diese nicht in der ersten Spalte einer Zeile stehen. Diese Position ist ausschließlich für Label reserviert.

Die Notation der Assemblerbefehle beim MSP430 folgt nicht der 3–Address–Notation. Vielmehr ist der erste Operand immer „src1“, während der 2. Immer sowohl „src2“ als auch „dst“ ist. Der im Beispiel gegebene Befehlt „mov #__STACK_END, SP“ (mov bedeutet „move“) würde in einer typischen 3–Address–Notation als „add #STACK_END, R0, SP“ geschrieben werden.

A.4 Erste Schritte in „C/C++“

C und C++ ähneln sich stark. Wenn Sie bereits in C++ programmiert haben, werden Sie feststellen, dass einige der Ihnen aus C++ bekannten Konstrukte in C nicht funktionieren. Umgekehrt ist die gesamte Syntax von C auch in C++ gültig. In diesem Versuch werden wir ausschließlich in C codieren.

In C ist es wichtig, das Konzept von Gültigkeitsbereichen und Blöcken zu verstehen. Beginnen Sie mit einer neuen Datei, so sind die folgenden 2 Arten von Statements möglich, um Code zu erzeugen bzw. Speicherplätze für Daten zu definieren.

- Datendefinition, Definition einer Variablen
  Sie folgen immer der Struktur „typname variablenname;“ Immer wenn eine Statement dieser Struktur im Quelltext auftaucht, so reserviert der Compiler den für den Datentype „typname“ notwendigen Speicherplatz und vermerkt den Namen „variablenname“ als Label für eben diese Adresse.
- Codedefinition

Alle Statements, die Operatoren enthalten, z.B. „+“+ oder „=“ (Zuweisung) dürfen nur innerhalb von Funktionen erfolgen. Eine Funktion erfolgt immer der Notation „typname funktionsname(parameterliste) { /* text */ }. Dabei ist eine Parameterliste entweder leer oder eine Liste, deren durch Kommata getrennte Elemente immer der Definition einer Variablen. An Stelle des Kommentars „/* text */ kommt nun der Code, der in der Funktion ausgeführt werden soll. Der Name der Funktion wird zu einem Label, dass die Adresse der ersten Anweisung der Funktion repräsentiert. Die geschlossenen geschweifter Klammer schließt die Funktion ab und lässt den Compiler an eben dieser Stelle eine „ret“ Statement erzeugen.

- Blöcke aus ‚{, und ‚}‘
  Paare aus geschweiften Klammern definieren in C (und C++) zusammenhängende Blöcke. In jedem Block kann zu Beginn die Definition von lokalen Variablen erfolgen. Dies gilt universell. Die hier definierten Variablen sind nur innerhalb des Blocks sichtbar, für den sie definiert sind. Sobald die schließende geschweifte Klammer des Blocks gelesen wurde, verlieren die inneren Variablendefinitionen ihre Gültigkeit.

  ACHTUNG: Seien Sie mit der Benennung von lokalen Variablen vorsichtig. Wenn Sie eine globale Variable definiert oder deklariert haben und dann in einem Block eine lokale Variable gleichen Namens definieren, so ist die globale Variable bis zum Ende des Blocks überdeckt und wird für den Code unsichtbar.

  ACHTUNG: Anders als in C++ ist es in C nicht erlaubt, lokale Funktionen zu deklarieren. Funktionsdefinitionen dürfen NUR auf der obersten Ebene der jeweiligen Quelldatei erfolgen.

A.5 Arbeit mit mehreren Quelldateien

Im Laufe des Praktikums wird eine einzelne Quelldatei mit hoher Wahrscheinlichkeit unübersichtlich werden. Es bietet sich dann an, mehrere Quelldateien in das Projekt einzufügen. Hierbei gilt, dass Sie alle „Deklarationen von Variablen“ oder allgemein gültigen Definitionen in so genannte „Header–Files“ (*.h) übertragen und diese dann, sofern verwendet, zu Beginn der jeweiligen *.c Datei mittels des „#include“ statements einbinden.

Achten Sie darauf, dass evtl. zugehörige „Definitionen“ genau einmal erfolgen. Die Deklaration einer Variablen erfolgt, indem Sie dem Text der Definition das Schlüsselwort „extern“ voranstellen. Die Deklaration einer Funktion erfolgt, indem Sie an Stelle der geschweiften Klammern mit dem Code nur ein einzelnes Semikolon nach der geschlossenen runden Klammer der Parameterliste schreiben.

A.6 Manipulation von Werten

In der Programmierumgebung eines Mikrocontrollers sind in der Regel alle wesentlichen Komponenten der MCU als Deklarationen verfügbar. Z.B, bezeichnet der Bezeichner „BIT0“ das nullte Bit in einem Byte. Auch die Namen der Register für die unterschiedlichen Hardwaremodule sind in den Standard–Headern der neu erzeugten Quelldateien passend vorhanden. Dies gilt ebenso für die Namen bestimmter funktionaler Bits oder Bitkombinationen in einem Register.

Bitte beachten Sie, dass die CPU Register immer nur parallel als Byte schrieben und Lesen kann. Wenn Sie den Wert einzelner Bits in einem Register lesen oder schreiben wollen, gehen Sie bitte wie folgt vor:

1. Lesen einzelner Bits
   Nutzen Sie den Operator „bitweises UND, &“ zur Auswertung. Der Teilausdruck P7IN & (BIT0 + BIT1) liefert einen Wert, in dem sicher alle Bits mit Ausnahme der Bits 0 und 1 auf „0“ gesetzt sind. Die beiden gesuchten Bits hingegen behalten die gelesenen Werte.
2. Explizites Setzen eines Bits
   Schreiben Sie „P7OUT |= BIT0“, um das Bit 0 im Register P7OUT zu setzen, ohne die anderen Bits des Registers zu beeinflussen. Hierbei ist der einzelne vertikale Strich der Operator „bitweises oder“.
3. Explizites Löschen eines Bits
   Schreiben Sie „P7OUT &= ~BIT0“, um das Bit 0 im Register P7OUT zu löschen, ohne die anderen Bits des Registers zu beeinflussen. Hierbei werden die bitweisen Operatoren ‚&‘ (und) und ‚~‘ (Negation) verwendet.
4. Explizites Invertieren von Bits
   Schreiben Sie P7OUT ^= BIT4 + BIT2, um die Bits 4 und 2 im Register P7OUT zu invertieren (umzuschalten), ohne die anderen Bits des Registers zu beeinflussen. Hierbei wird der einzelne der Operator „Caret, ^“, bitweises exklusiv–oder“, angewendet.

Bitte beachten Sie, dass bei einer einfachen Zuweisung zu einem Register oder einem beliebigen Wert immer ALLE Bits in diesem Wert manipuliert werden!

A.7 Arbeiten mit bedingter Compilierung

C (und auch C++) erlauben es, einen Quelltext so zu gestalten, dass Modifikationen in Abhängigkeit vom Einsatzweck automatisiert möglich sind. Hierzu werden Quelltextstücke durch Compileranweisungen wie

#ifdef

#else

#endif

strukturiert. Das CCS macht von dieser Möglichkeit exzessiv Gebrauch, um die die gemeinsamen Definitionen der MSP430 Module auf die konkrete, verwendete CPU anzupassen.

Die Symbolik dieser Compiler-Direktiven hängt immer mit bekannten Symbolen zusammen. Letztere finden sich entweder direkt in Ihrem Quelltext, z.B. wie

```
#define MySymbol
```

Nach dieser Zeile kennt der Compiler das Symbol „MySymbol“ und wird alle Blöcke, die mit der Kombination

```
#ifdef MySymbol
/*** hier kommt bedingter Code hin **//
#endif
```

umschlossen sind, mit in den erzeugten Code aufnehmen.

Eine andere, elegantere Möglichkeit, diese Definitionen festzulegen, ist ein Eintrag in die Projektoptionen Ihres C/C++ Projekts, wie in Abbildung 8 dargestellt.**Abbildung 1**

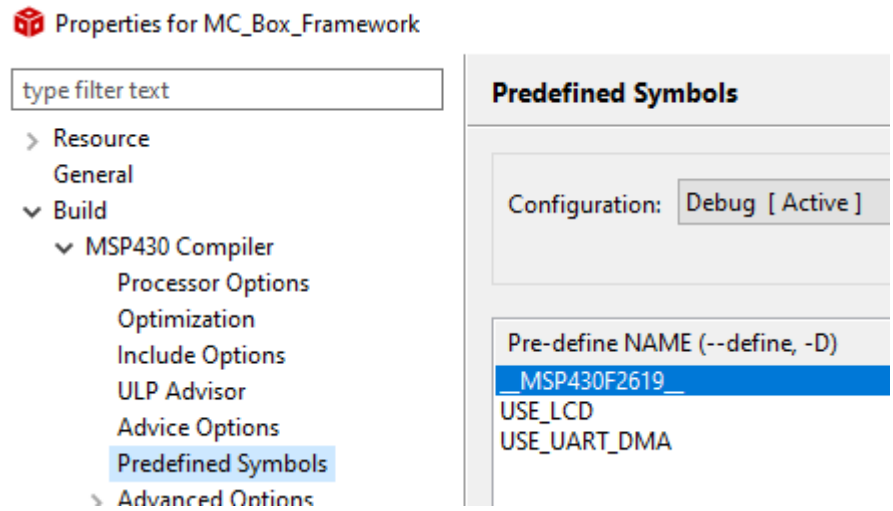

Abbildung 8: Definition von Symbolen für den Compiler

Für den in diesem Praktikum erzeugten Code macht so ein Vorgehen z.B. Sinn, um Datentypen und ganze Softwareblöcke auf unterschiedliche Implementierungen umzustellen. (Siehe auch Anhang B oder die Implementierung der LED-Steuerung wahlweise per Interrupt oder DMA)

Anhang B, Datenstrukturen für einzelnen Probleme

B.1 Datenstruktur für die LED-Reihe

Je nachdem, ob Sie für die Übertragung an die LED-Reihe MSB- oder LSB-First wählen, ändert sich die Datenstruktur, die Sie im Speicher ablegen müssen. Der folgende Code findet sich auch in einem der Quelltext-Schnipsel in Moodle wieder.

a) LSB-First

```
typedef struct _tWS_Code_LSB {  // one byte of color data for the LED
WS2812B will be represented as a 3-byte entity which collects
    // all the values along with all the start- and stop steps.
    union {   // all elements on this level will overlay physically
        struct {  // this will be the bitwise description. The SPI is
supposed to send this in ascending order, LSB first
            uint8_t : 1;    // anonymous start bit, accessible only via
the byte representation
            uint8_t b7: 1;
            uint8_t : 2;    // one stop plus one start bit
            uint8_t b6 : 1;
            uint8_t : 2;
            uint8_t b5: 1;
            uint8_t : 2;
            uint8_t b4 : 1;
            uint8_t : 2;
            uint8_t b3 : 1;
            uint8_t : 2;
            uint8_t b2 : 1;
            uint8_t : 2;
            uint8_t b1 : 1;
            uint8_t : 2;
            uint8_t b0 : 1;
            uint8_t : 1;      // the last stop bit
        };
        struct {
                 // this will be the byte-wise description
            uint8_t B0;
            uint8_t B1;
            uint8_t B2;
        };
    };
} TWSCode_LSB;
```

b) MSB-First

```
typedef struct _tWS_Code_MSB {
    union {
        struct {
            uint8_t b5 : 1;            // notation is from LSB to MSB
            uint8_t : 1;
            uint8_t : 1;
            uint8_t b6 : 1;
            uint8_t : 1;
            uint8_t : 1;
            uint8_t b7 : 1;
            uint8_t : 1;


            uint8_t : 1;
            uint8_t : 1;
```

```
            uint8_t b3 : 1;
            uint8_t : 1;
            uint8_t : 1;
            uint8_t b4 : 1;
            uint8_t : 1;
            uint8_t : 1;


            uint8_t : 1;
            uint8_t b0: 1;
            uint8_t : 1;
            uint8_t : 1;
            uint8_t b1 : 1;
            uint8_t : 1;
            uint8_t : 1;
            uint8_t b2 : 1;
        };
        struct {
            uint8_t B0;
            uint8_t B1;
            uint8_t B2;
        };
    };
} TWSCode_MSB;
```

In diesen Beispielen werden fortgeschrittene Techniken der C/C++ Nomenklatur verwendet:

a. Bitfelder
   Es ist möglich, ganzzahlige Datentypen in einer Struktur aufzulisten und die Länge des einzelnen Feldes explizit anzugeben. In den zuvor gezeigten Datenstrukturen sind dies Einzelbitfelder! (: 1). Es wird zwischen benannten (mit eindeutigem Elementnamen) und unbenannten (ohne Elementnamen) Feldern unterschieden. Bei der Verwendung einer Variablen des Datentyps oder eines Zeigers darauf sind dann ausschließlich die benannten Felder sichtbar. Die unbenannten Teile lassen sich nicht modifizieren.
   Bei der Zuweisung von Werten auf eines dieser Felder erzeugt der Compiler automatische die korrekten Masken und Verknüpfungen.
   Wichtig zu wissen ist, dass der Compiler die Felder in aufsteigender Wertigkeit zählt. D.h. das erste Feld der Struktur belegt die niederwertigsten Bit im angefangenen Byte!
b. „union“ Datentypen.
   Eine „Union“ sieht formell einer Struktur ähnlich. Allerdings referenziert jedes Elemeten der „Union“ den gleichen Speicherplatz. So wird es möglich, ein und dasselbe Feld im Hauptspeicher differenziert zu interpretieren. Im obigen Beispiel überlagern sich 2 Strukturen. Zum einen ein komplexes Bitfeld von 24 Bit, zum anderen eine einfache Struktur aus 3 Byte. Dies wird in er Implementierung (siehe Beispiel-Quelltexte) genutzt, um über die 3-Byte Struktur eine Initialisierung aller Bits im Speicher vornehmen zu können, später aber nur noch über die Bitfelder einzelne Bits zu modifizieren.