Universität Kassel
Fachbereich Elektrotechnik/Informatik
Prof. Dr. B. Sick
Fachgebiet Intelligente Eingebettete Systeme

UNIKASSEL VERSITÄT
Intelligent Embedded Systems

Labor C/Embedded Systems

Teil 2

Intelligente Eingebettete Systeme

Aufgabenblatt 3

Bluetooth, Spannungsmessung, Ultraschall, Servomotor, Softwaregestaltung

Betreuer: Benjamin Herwig <bherwig@uni-kassel.de>
und viele großartige Tutoren

Aufgabe 1: Bluetooth – Serielle Kommunikation per Funkstrecke

Die Beinchen `PD0` und `PD1` des Microcontrollers sind mit `RX` und `TX` (dabei handelt es sich um *Sonderfunktionen*, eben zur seriellen Kommunikation, des Microcontrollers) beschriftet. Diese Beinchen ermöglichen die serielle Kommunikation von anderen seriellen Geräten mit dem Microcontroller. Das mit `RX` beschriftete Beinchen wird zum seriellen Empfang (*Receive*) genutzt, das mit `TX` beschriftete Beinchen zum seriellen Senden (*Transmit*).

Auf dem Roboter ist ein Bluetooth-Modul verbaut, das ebenfalls über eine Am Bluetooth-Modul gibt es auch `RX`- und `TX`-Anschlüsse. Diese müssen „über Kreuz“ mit dem Microcontroller verbunden werden, d. h. es müssen die Verbindungen `RX`–`TX` und `TX`–`RX` hergestellt werden.

Zum Herstellen der entsprechenden Verbindung verfügt Ihr Roboter über ein zweiadriges, grau-weißes Kabel, das an den passenden Eingängen am Bluetooth-Modul fixiert ist. Das andere Kabelende ist steckbar ausgelegt und kann auf die Pins, die auf dem Shield mit `RXD` und `TXD` bezeichnet sind, gesteckt werden.

Wenn Sie das freie Kabelende so stecken, dass die weiße Ader auf `TXD` und die graue Ader auf `RXD` liegen, kommt die gekreuzte Verbindung wie oben genannt zustande.

Sie können keinen Programmcode über die Bluetooth-Verbindung flashen! Falls Sie Programmcode auf den Microcontroller flashen wollen, kann es zu Problemen kommen, falls diese Hardware-Verbindung hergestellt ist. Lösen Sie sie notfalls.

Seien Sie vorsichtig beim Stecken und Lösen. Stecken Sie das freie Kabelende niemals auf andere Pins als die genannten. Stecken Sie das freie Kabelende auch niemals nur teilweise auf – die Verbindung gilt dann nicht als gelöst (und auch nicht unbedingt als hergestellt)!

Das Bluetooth-Modul wird kontinuierlich mit einer Spannung versorgt, sowohl beim Betrieb des Roboters per USB-Kabel als auch im Akkubetrieb. Um das Anschalten des Bluetooth-Moduls brauchen Sie sich also nicht kümmern. Das Bluetooth-Modul ist für den Microcontroller *transparent*. Vom Microcontroller aus können Sie an beliebige serielle Geräte Daten senden (oder von ihnen empfangen). Es macht also keinen Unterschied, ob Sie mit dem Roboter per USB-Kabel oder per Bluetooth-Verbindung reden wollen. Vielmehr ist es so, dass die `RX`- und `TX`-Beinchen des Microcontrollers *parallel* sowohl mit dem USB-Anschluss (über ein bisschen dazwischengehängte Eleketronik) des Arduino-Trägerboards als auch mit dem Bluetooth-Modul verbunden sind.

Nachdem die Hardware-Verbindung der seriellen Leitungen auf dem Roboter hergestellt wurde, muss nun also Rechner-seitig eine Verbindung hergestellt werden. Sie kennen bereits das Vorgehen, sich mit der Datei `/dev/ttyACM0` zu verbinden, die im `/dev`-Verzeichnis Ihrs Rechners angelegt wird, sobald Sie das USB-Kabel in den Roboter und in den Rechner eingesteckt haben. Diese Datei repräsentiert sozusagen die serielle Verbindung mit der Arduino-Elektronik.

Eine solche Datei müssen Sie entsprechend auch für die Bluetooth-Verbindung herstellen. Die Labor-Rechner im FG IES verfügen über ein Bluetooth-Modul. Dieses Bluetooth-Modul muss mit einem Tool, das `rfcomm` genannt wird, angesprochen werden. Wenn das Tool korrekt verwendet wird, wird eine Datei mit dem Namen `rfcomm0` im `/dev`-Verzeichnis Ihres Rechners angelegt. Diese Datei können Sie dann beispielsweise mit `cutecom` zur Kommunikation mit Ihrem Roboter öffnen, so wie Sie das schon kennengelernt haben.

Die Verwendung von `rfcomm` wird für Sie in einem Script erledigt, das auf den Rechnern zur Verfügung steht und per Kommandozeile genutzt werden kann. Öffnen Sie ein Terminal und geben Sie `BotBtSerial` (oder einfach `Bot`, gefolgt von der Tabulatortaste) ein. Sie erhalten eine Fehlermeldung, dass das Script einen Parameter benötigt. Zum Verbinden eines Roboters wird der parameter `connect` verwendet. Geben Sie also `BotBtSerial connect` ein. Danach wird für einen Moment – bis zu ungefähr eine Minute kann das dauern – nach Robotern gesucht, deren Bluetooth-Modul aktiv ist. In der Liste der zur Verfügung stehenden Roboter wählen Sie – immer nur – Ihren Roboter aus und geben die Ihrem Roboter zugeordnete Nummer aus der Liste ein.

Nun wird die Gerätedatei `rfcomm0` angelegt und Sie können sich in Cutecomn über diese Datei per Bluetooth mit Ihrem Roboter unterhalten.

Verifizieren Sie, dass alles geklappt hat, indem Sie ein kleines Microcontroller-Programm schreiben, dass Ihren Vor- und Zunamen per serieller Schnittstelle sendet. Flashen Sie das Programm, und *lösen Sie dann die USB-Verbindung*. Legen Sie

Sollten Sie Ihren Roboter per Batterien betreiben und sie ihn am Schiebeschalter ausschalten und außerdem vom USB-Kabel gelöst haben, schaltet sich sämtliche Elektronik des Roboters aus. Dann bricht natürlich auch die Bluetooth-Verbindung zusammen und die Datei `rfcomm0` wird gelöscht. Dann müssen Sie erneut eine Verbindung herstellen, natürlich per `BotBtSerial`.

Tipp: Schreiben Sie Informationen per USART „nach draußen“ um das Verhalten Ihres Roboters besser analysieren zu können. Bedenken Sie aber, dass Kommunikation auch immer zeitaufwendig ist. Implementieren Sie also beispielsweise eine Debug-Komponente in Ihren Code, die Sie – vielleicht ja per Preprocessor-Flag – deaktivieren oder sogar bzgl. ihrer *Verbosity* einstellen können.

Aufgabe 2: Analogmessung der Linienfolgerspannung

Die Linienfolgerelemente (links, mittig, rechts) können statt der digitalen 0 oder 1, also `LOW` oder `HIGH`, auch analoge Spannungspegel ausgeben. Auch diese können vom Microcontroller gemessen werden, und zwar im Bereich von 0 Volt bis ein bisschen weniger als `AREF` Volt, wobei es sich bei letzterem um eine Spannungsmessungs-Referenzspannung handelt. Auf unserem Microcontroller können wir $\texttt{AREF} = \texttt{VCC}$ annehmen (wobei diese bei Akkubetrieb recht genau 5 Volt beträgt, bei USB-Betrieb ein bisschen weniger). Eine Referenzspannung im Microcontroller dient dazu, andere Spannungen mit ihr zu vergleichen. Beispielsweise kann die Analogspannung der Linienfolgerelemente damit vergleichen werden/ In der Regel ist die maximal an der Kollektor-Emitter-Strecke abfallende Spannung etwas geringer als `VCC`, aber ein wenig größer als `GND`, da es sich nicht um ideale Bauteile handelt (was im echten Leben quasi nie der Fall ist).

Die Spannungswerte werden mit 10 Bit kodiert, d. h, dass in dem o. a. Spannungsbereich 1024 verschiedene Spannungen gemessen werden können, der Wert `VCC` (bzw.ein bisschen weniger) entspricht dabei dann dem höchsten von uns messbaren Spannungswert.

Sollte der IR-Fototransistor auf einem Linienfolgerelement also „zur Hälfte“ durchgesteuert sein, dort also (ein bisschen weniger als) $\frac{\texttt{VCC}}{2}$ abfallen, würden wir den Wert $1023/2$ messen, wenn wir ideale Bauteile hätten. (In Wahrheit fällt noch ein klein wenig Spannung an einem in Reihe geschalteten Widerstand ab, der mit dem Fototransistor einen Spannungsteiler bildet – das spielt für das konzeptuelle Verstehen hier erstmal keine Rolle. Schauen Sie ins Skript!)

Die Spannungsmessung an unserem Microcontroller kann über die Analog-Eingänge `PC[0,1,2(,3)]`) erfolgen; `PC3` kann für die Messung der Akkuspannung verwendet werden – experimentieren Sie gerne auch damit.

Die Eingänge (`PC[0|2]`) wurden von uns bisher als digitale Eingänge genutzt.

Um die Linienfolgerausgangsspannung messen zu können, müssen die entsprechenden Beinchen weiterhin jeweils als Eingang konfiguriert werden.

Der Microcontroller muss nun jedoch so konfiguriert werden, dass die Beinchen im inneren des Microcontrollers mit dem Analog-Digital-Umsetzer (*ADU* oder *ADC* für Analog-Digital-Converter) verbunden werden. Dafür werden die beiden Register `ADMUX` und `ADCSRA` benötigt.

Im Register `ADMUX` wird ausgewählt, auf welchem Kanal unseres Microcontrollers wir die Spannung messen wollen. Ein Kanal entspricht einem Microcontrollerbeinchen (`PC[0...5]`).

An `PC[0...2]` sind die Analogausgänge der Linienfolgerelemente angeschlossen.

Je nach Reflektionsgrad können die Linienfolgerelemente einen Spannungswert zwischen 0 Volt und (fast) 5 Volt ausgeben, der gemessen werden kann.

Im Register `ADMUX` muss ein Wert für eine *Referenzspannungs (Vergleichsspannung)* definiert werden. Das geschieht dadurch, dass man über entsprechende Bits auswählt, von welcher Stelle der Microcontroller die Referenzspannung beziehen soll. Durch das Setzen von `REFS0` auf 1 und `REFS1` auf 0 wird die Versorgungsspannung des Microcontrollers als Referenzspannung ausgewählt (Achtung: Dabei handelt es sich *nicht* um die Batterispannung – die Batteriespannung wird über einen *Step-Down-Regler* auf dem Arduino-Board auf (knapp unter) $5V$ geregelt, was dann der `VCC` des Microcontrollers entspricht, was wiederum das gleiche ist wie `AVCC`.)

Außerdem wird in `ADMUX` ausgewählt, welcher der Kanäle ausgelesen werden soll. Dazu sind die Bits `MUX[0...3]` vorgesehen. Ein Kanal entspricht dabei einem Anschlussbeinchen. Die folgende Tabelle gibt die Bitbelegung für das jeweilige Anschlussbeinchen bzw. den entsprechenden Kanal an:

MUX3	MUX2	MUX1	MUX0	Kanal
0	0	0	0	PC0
0	0	0	1	PC1
0	0	1	0	PC2

Es genügt hier also, den dezimalen Wert 0, 1 oder 2 an die 4 Bitstellen „ganz rechts“ im `ADMUX`-Register zu schreiben.

Bedenken Sie aber, dass Sie, wenn Sie einen Kanal auswählen, die Konfigurationsbits für die Referenzspannungsquelle nicht (mit etwas Unpassendem) überschreiben. Setzen Sie außerdem niemals das `ADLAR`-Bit.

Ein weiteres zur Analog-Digital-Umsetzung benötigtes Register ist `ADCSRA`. In diesem Register wird eine Analog-Digital-Umsetzung erlaubt (durch setzen des Bits `ADEN`) und gestartet (durch setzen des Bits `ADSC`). Außerdem wird in diesem Register durch das sinnvolle Belegen der Bits `ADPS[0...2]` eine Taktvorteilung ausgewählt, mittels derer die Anzahl der Messvorgänge pro Sekunde angegeben werden kann. Diese Frequenz sollte zwischen 50 und 200 kHz (Kilohertz) liegen.[1]

Die Samplingfrequenz wird über eine Taktovorteilung ausgewählt, und diese muss bei uns 128 betragen, so dass $125kHz$ als Samplefrequenz genutz werden werden. Hierzu müssen alle Bits `ADPS[0...2]` auf 1 gesetzt werden.

Sobald die Konfiguration der Analog-Digital-Wandlung abgeschlossen ist, kann eine ebensolche gestartet werden. Hierzu muss in `ADCSRA` das Bit `ADSC` (`SC` für *Single Conversion*) gesetzt werden. Danach startet eine AD-Wandlung. Ein solcher Vorgang kostet immer Zeit (denken Sie an die $125kHz$), bedingt durch die Art der AD-Umsetzung.[2]

Sobald die AD-Umsetzung abgeschlossen ist, wird das Bit `ADSC`, das wir ja manuell auf 1 gesetzt haben, vom Microcontroller auf 0 gesetzt. Das bedeutet, dass wir (z. B.) prüfen können, ob die Umwandlung abgeschlossen wurde, indem wir prüfen, ob `ADSC` gelöscht ist. Hierzu kann man z. B. eine Warteschleife nutzen.[3]

Nachdem eine Wandlung abgeschlossen wurde, kann das Wandungsergebnis gelesen werden. Da es sich um einen 10-Bit-Wandler handelt, müssen wir einen 10-Bit-Wert behandeln. Der Microcontroller ist jedoch ein 8-Bit-Microcontroller und kann aus seinen Registern nur 8 Bit lesen (da die Register nur 8 Bit breit sind). Es können generell keine Daten breiter als 8 Bit innerhalb einer Instruktion verarbeitet werden (jedoch können

[1] Warum? Erfahren Sie in unserer Veranstaltung *Intelligente Technische Systeme*.

[2] Mehr in ... ITS!

[3] Jedoch ist auch hier (natürlich) eine Behandlung per Interrupt möglich! Schauen Sie ins Datenblatt! Aber eine gepollte Wandlung ist in unserem Fall schnell genug und sehr viel einfacher umzusetzen.

längere Adressen genutzt werden, aber das ist ein anderes Thema – der Adressbus des Microcontrollers ist breiter als 8 Bit, der Datenbus aber nicht).

Man behilft sich derart, dass das Wandlungsergebnis über zwei Register verteilt wird. Die niederwertigen 8 Bit des Wandlungsergebnisses stehen im Register `ADCL`, die hochwertigen (2 Bit) im Register `ADCH`. Man kann diese Register nun einzeln lesen (und sie durch Shiftoperationen in einem 16 Bit Datentyp darstellen), es gibt jedoch auch ein Makro, dass das bereits übernimmt. Hierbei handelt es sich um das Makro `ADCW`, wobei das `W` für *wide* oder *whole* steht. `ADCW` liefert also einen 16 Bit Wert zurück. Aber dieser ist erst dann valide (natürlich mit einem gewissen Messfehler der für uns in dieser Veranstaltung als irreduzibel gilt[4]), wenn gilt, dass `ADSC == 0`.

Außerdem muss der Analog-Digital-Umsetzer kurz „warmlaufen“. Die allererste ADC-Messung nach einem (Neu-) Start des Microcontrollers ist nicht aussagekräftig. Sie sollte nicht weiterverwendet werden. Man kann also die Register einfach auslesen und nichts weiter mit dem Resultat tun.

In der Datei `iesadc.c` im moodle finden Sie das entsprechende Vorgehen. Um die Datei als Programm nutzen zu können müssen Sie auch die Funktionalität der seriellen Schnittstelle mit einkompilieren.

Nutzen Sie die ADC-Funktionalität z. B. dafür, sich Ihrer Linienfolgerzustände sicherer zu werden.

Aufgabe 3: Zeitmessung mit Interrupts

Um mit dem Microcontroller eine Zeitspanne zu messen, verwenden wir einen der auf dem Microcontroller verbauten Timer bzw. Zähler. `Timer1` ist ein 16-Bit-Zähler, der also 65536 unterschiedliche Zählerstände (in einem Register des Microcontrollers) annehmen kann. Der Zählerstand wird entweder mit dem Systemtakt oder mit dem vorgeteilten Systemtakt (per *Taktvorteiler*) hochgezählt (und stets um 1 inkrementiert). Je nach Zählerstand können unterschiedliche Aktionen, sogenannte *Interrupts*, ausgelöst werden.

Wenn ein Interrupt ausgelöst wird, wird eine bestimmte, diesem Interrupt zugeordnete *Interrupt Service Routine (ISR)* ausgeführt. Interrupt Service Routinen sind mit Methoden vergleichbar, haben jedoch keine Rückgabe. Sie werden, falls sie erlaubt wurden, während des Programmablaufs beim Auftreten eines Interrupt-Signals angesprungen und durchlaufen, und zwar ganz egal, an welcher Stelle der Programmablauf sich gerade befindet.

Da also während der Behandlung einer ISR die Ausführung des eigentlich laufenden Programms an einer unbestimmten Stelle unterbrochen wird, ist es wichtig, so wenig Anweisungen wie möglich in einer ISR unterzubringen, damit sie schnell endet und an die Stelle des ursprünglichen Programmablaufs zurückgekehrt wird.

Interrupts können beispielsweise dann ausgeführt werden, wenn der Zählerstand eines Timers dem Wert entspricht, der in einem Vergleichsregister abgelegt wurde – hierbei handelt es sich dann um einen sog. *Compare-Match-Interrupt*.

In einer ISR, die z. B. einem Compare Match des Timer1 zugeordnet ist, können z. B. Zählervariablen im C-Programm hochgezählt werden (die dann global sein dürfen). Dadurch, dass man weiß, wie oft pro Sekunde der Zähler inkrementiert wird (da die Taktgeschwindigkeit des Microcontrollers bekannt ist) und auch, wie oft ein Compare Match ausgeführt wird, kann man aus den Werten der Zählerstände verstrichene Zeit ableiten. Hierbei ist jedoch zu beachten, dass (sehr wenige) Mikrosekunden verbraucht werden, in denen die ISR aufgerufen und durchlaufen wird. In einfachen Anwendungen ist das jedoch zu vernachlässigen (aber dennoch ist es stets gut, den zu erwartenden Fehler zu quantifizieren… dazu müsste man sich den dem C-Code zugrunde liegenden Assembler-Code anschauen).

Im moodle-Kurs finden Sie einen C-Quelltext mit dem Titel `iesisr.c`. In diesem finden Sie ein Beispiel, in dem ein Register, das Timer 1 zugeordnet ist, mit einem Wert befüllt wird, bei dessen Erreichen ein dem Timer 1 zugeordneter Compare-Match-Interrupt ausgelöst wird, mit dem eine globale Variable inkrementiert wird.

Nutzen Sie diesen Quelltext als Grundlage für Zeitmessungsaufgaben.

[4] Vgl. ITS!

Aufgabe 4: Software-Gestaltung, Coding-Conventions

Sie haben mittlerweile schon ein bisschen Quelltext geschrieben und gewiss bemerkt, dass man mit den „Beinchenkonstanten“ schnell durcheinander kommt. In diesem Kontext wurde Ihnen schon ein Notationsvorschlag in der Beispiellösung zur Schieberegister-Aufgabe unterbreitet.

Außerdem steigt die Komplexität eines Programmcodes (oft) mit seiner Länge, und dadurch auch die Fehleranfälligkeit.

Nutzen Sie die sprachlichen Mittel, die Ihnen die Programmiersprache C bietet, um Ihren Code möglichst sinnvoll zu gestalten.

Dazu gehört z. B. die Verwendung von

- (Präprozessor-) Konstanten und Präprozessor-Makros,
- Enumerationen,
- Funktionen und Methoden,
- „sprechenden“ Variablen- und Funktionsnamen,
- sinnvollen Kommenaren (unterschiedliche Kommentar-Arten in Header-Dateien und Implementierungen!) und
- unbedingte Vermeidung von Code-Redundanzen (die teilweise nur durch die Nutzung von Präprozessor-Makros möglich ist).

Zur Orientierung, wie C-Quelltext „schön“ aussehen kann, sollten Sie sich an dem Dokument `Coding-Conventions.pdf` im moodle orientieren.

Eine Einhaltung der Coding-Conventions ist Bestandteil der Benotung am Ende des Praktikums (zu dem Thema später mehr). Bis zum Ende der Lehrveranstaltungszeit müssen Sie evtl. noch mit Änderungen der Coding-Conventions rechnen; Änderungen werden ggf. im moodle bekannt gegeben.

Machen Sie sich klar, dass es einen großen Unterschied macht, wo Sie die grunsätzliche Schnittstellendokumentation des Softwareprojekts dokumentieren. Beispielsweise könnten Sie ja versucht sein, ein Schieberegister-Modul zu entwickeln, das Sie nur in Form von Header-Datei und Objektdatei, jedoch ohne C-Implementierungsquelltext verteilen. Die Dokumentation der Schnittstellenfunktionen sollte aber natürlich auch im Quelltext mit herausgegeben werden, so dass Ihr *Kunde* oder ein *Betreuer* jederzeit aus dem Gesamtprojekt, in dem Ihr Modul mit Ihrer Header-Datei verwendet wird, eine neue Dokumentation erzeugen kann, die *auch* die von Ihnen stammende Schnittstellendokumentation enthält. Also nehmen Sie folgende Regel mit: Schnittstellendokumentation gehört in Headerdateien. Nur Implementierungshinweise, wobei es sich in der Regel um sehr kurze Kommentare handelt, gehören in die C-Implementierung. Außerdem gehört zu (so gut wie) jeder C-Datei auch eine Header-Datei.

Wahrscheinlich müssen Sie auch damit rechnen, dass Sie im Rahmen der Abschlussaufgabe dazu gezwungen(!) werden, Ihre Variablenbezeichner nach einer Konvention zu benennen, die sich *ungarische Notation* nennt. Recherchieren Sie schonmal, was das ist, und auch, warum sie sehr oft zu Unrecht kritisiert wird (stellen Sie sich in diesem Zusammenhang die Frage: Was ist der *Typ* einer Variable, also was verbirgt sich hinter dem Begriffskonstrukt des *Typs einer Variable*. Beantworten Sie die Frage nicht mit Mitteln aus der theoretischen Informatik, sondern überlegen Sie, welchen *Aufgabentyp* eine Variable hat bzw. welchen Aufgabentyp der Bezeichner einer Variable beschreibt, und setzen Sie den *Aufgabentyp* in Verbindung mit dem, was Sie zur ungarischen Notation recherchieren.)

Aufgabe 5: Quellcode kommentieren und *dokumentieren*

Um sich einen Überblick über ein Softwareprojekt zu verschaffen – besonders: um die Frage zu beantworten, wie ein Softwareprojekt *genutzt* werden kann – ist eine ausführliche und sinnvolle Dokumentation nötig.

Insbesondere wenn man bedenkt, dass viele Softwareprojekte aus sehr vielen (C-) Modulen bestehen, ist klar, dass es nicht genügt, z. B durch den ganzen Quelltext zu lesen, um zu sehen, was wo wie ausgelöst werden kann.

Um jedoch auch nicht „zwei Arten Dokumente“ zu pflegen, ist es sinnvoll, die Dokumentation weitgehend automatisch zu erledigen, beispielsweise im Quelltext direkt (dort per Kommentaren).

Das Werkzeug `doxygen` unterstützt Sie beim Dokumentieren Ihres Software-Projekts.

Lesen Sie das Dokument `Doxygen-Tutorial.pdf` (moodle) um ein wenig in die Doxygen-Thematik rein zu kommen; die im Essay besprochenen Dateien liegen im `.zip`-File mit dem Namen `Doxygen-Tutorial-Dateien.zip`.

Generieren Sie dann ein Doxyfile und passen Sie den Eintrag `OPTIMIZE_OUTPUT_FOR_C` an. Sorgen Sie außerdem dafür, dass die Rümpfe Ihrer Funktionen im Dokument mit angezeigt werden. Schauen Sie außerdem, wie Sie mittels *Markdown*[5]-Syntax Ihrer Dokumentation aufhübschen können.

Nutzen Sie zum Anpassen der Dokumentation nicht den GUI-Wizard, sondern arbeiten Sie direkt im Doxyfile (z. B. mit `geany` oder `nano`).

Gut ist es erstmal, wenn Sie darauf achten, dass Sie Ihr Doyfile in dem Verzeichnis erzeugen, in dem auch die zu dokumentierenden Quelltexte liegen.

Für diese Aufgabe gilt: Selber machen! Recherchieren Sie, wie man das Tool verwendet. Z. B. auch so: `http://lmgtfy.com/?s=d&q=doxygen+c`

Tipp: Sie können und müssen fehlende Werkzeuge im virtuellen Labor selbst installieren. Zum Installieren von Werkzeugen im virtuellen Labor beachten Sie bitte, dass das root-Passwort `ies` lautet. Als root-Benutzer kann z. B. `splint` so installiert werden: `apt install splint`.

Aufgabe 6: Quellcode automatisch *verschönern* lassen

Wie Sie schon in der Vorlesung gelernt haben, ist es sinnvoll, Quelltext ordentlich zu strukturieren, also vernünftig einzurücken usw.

Für diejenigen unter Ihnen, die das immer noch nicht glauben und deswegen davon absehen: Es gibt ein Kommandozeilenwerkzeug, das Ihnen hilft. `(GNU) indent`.

`indent` ist auf den Laborrechnern installiert. Nutzen Sie es, um sich mal verschiedene Einrückungsstile anzuschauen. `indent` arbeitet per Standardeinstellung mit dem `gnu`-Stil. Sorgen Sie jedoch dafür, dass die Funktionsrümpfe in der Doxygen-Dokumentation gemäß `K&R` formatiert sind. Sie müssen also vor dem Benutzen von Doxygen Ihre Quelltexte anpassen (notfalls händisch, aber am besten mit `indent`).

Tipp: `indent` müssen Sie auch installieren. Das Debian-Paket, das `indent` zur Verfügung stellt, heißt `indent`.

Sehr wichtiger Tipp: In der Doxygen-Dokumentation sollte das Dokumentationswerkzeug auch den Quelltext als Textbaustein(e) einfügen. Lassen Sie `indent` daher unbedingt *vor* dem Erzeugen der Dokumentation laufen. Beachten Sie auch die folgende Aufgabe zu `make`.

Für diese Aufgabe gilt: Selber machen! Recherchieren Sie, wie man das Tool verwendet. Z. B. auch so: `http://lmgtfy.com/?s=d&q=gnu+indent`

Aufgabe 7: Automatisierung von Buildprozessen – Make

Im Verlauf der Entwicklung Ihrer Robotersteuerung werden Sie immer mehr Module anlegen, um die Funktionalität auf diese zu verteilen und Ihren Programmcode sinnvoll zu strukturieren und zu kapseln.

Wie Sie wissen, können Sie C-Module, also einzelne `.c`-Dateien, separat kompilieren (und dadurch Objektdateien erzeugen), um diese in einem weiteren Schritt zu *linken*.

Normal ist, dass man im Verlauf einer Softwareentwicklung eine Reihe von Dateien bearbeitet, andere aber nicht – möglicherweise arbeiten sie wochenlang an der Linienfolgealgorithmik, ändern aber nichts an der auch in Ihrem Projekt verwendeten USART-Implementierung. In diesem Fall ist es natürlich sinnvoll, nur die veränderten Module zu kompilieren, die unveränderten jedoch nicht (sondern die Objektdateien vorzuhalten und direkt zu linken).

[5] Das zu kennen lohnt sich auf jeden Fall!

Das spart sehr viel Rechenzeit, da (sehr viele große) C-Projekte mitunter sehr lange (Tage!) zum Kompilieren brauchen.

Ein Werkzeug, das die Ahängigkeiten in Softwareprojekten automatisiert verwalten und prüfen kann, ist `make`. `make` wird über ein `makefile` konfiguriert; dort sind die Abhängigkeiten, die es in einem Softwareprojekt gibt, modelliert. Beispielsweise setzt ein Projekt voraus, dass USART verwendet wird. Sollte es noch keine USART-Objektdatei geben, oder diese älter sein als der dieser Datei zugrunde liegende Quelltext, so wird diese Objektdatei erzeugt.

Lesen Sie das im moodle bereitgestellten Erklärtext zu Make, um ein wenig in die Make-Thematik hinein zu kommen.

Überlegen sie dann, welche unterschiedlichen Targets in Ihrem Softwareprojekt Sinn ergeben; was wird alles getan (und ist z. B. in `compile.sh` in einen Schritt zusammengefasst)? Erstellen Sie ein einfaches Makefile mit sinnvollen Targets.

Recherchieren Sie auch, was ein sog. `phony`-Target ist und wann es Sinn macht.

Ihr Makefile sollte mindestens die Targets `compile`, `flash`, `link`, `clean`, `.PHONY`, `documentation` und `all` enthalten. Im Rahmen der Abschlussaufgabe sollte Ihr Makefile noch Targets zum *linten* und zum Erzeugen der Dokumentation enthalten.

Entwickeln Sie Ideen für weitere Targets.

Sie können `make` aus der Konsole und auch aus `geany`[6] heraus benutzen. (Viele Entwicklungsumgebungen nutzen im Hintergrund `make`. Codeblocks aber nutzt per Standardeinstellung ein anderes, eigenes Build-System, das Sie aber bitte nicht benutzen werden. Falls Sie mit Codeblocks programmieren, können Sie auch make verwenden, auch wenn es nicht in Codeblocks integriert ist.)

Sie müssen `make` in der Abschlussaufgabe benutzen und ein Makefile mit Ihrem Programmcode abliefern. Dieses Makefile muss dann die Roboter-Firmware erzeugen, das Flashen übernehmen, die Dokumentation erstellen und vieles mehr.

Generelle bei Fehlern zu bedenkende Dinge:
Auftretende Fehler, die Ihnen am Anfang vielleicht das Leben in diesem Praktikum schwer machen sind beispielsweise:

- Sind alle Kabel korrekt verbunden?
- Ist der Servo mechanisch blockiert? Falls Sie diesen Eindruck haben, wenden Sie sich an die Betreuenden.
- Haben Sie die selben Timer für unterschiedliche Aktionen verwendet? Bedenken Sie, dass es drei Timer auf dem Microcontroller gibt und dass jeder Timer über jeweils zwei Compare-Match-Register verfügt.
- Falls eine seltsame Entfernung mit dem Ultraschallsensor gemessen wird: Welche Gestalt hat das reflektierende Objekt und aus welchem Material besteht es? Beides hat Auswirkungen auf die Reflexionseigenschaften. Außerdem könnte der Fall auftreten, dass es zu Berechnungsproblemen kommt, sodass Sie plötzlich eine negative Burst-Laufzeit ermitteln. Treffen Sie entsprechende Vorkehrungen.

[6] Schauen Sie mal ins Menü, befragen Sie auch die `geany`-Dokumentation.