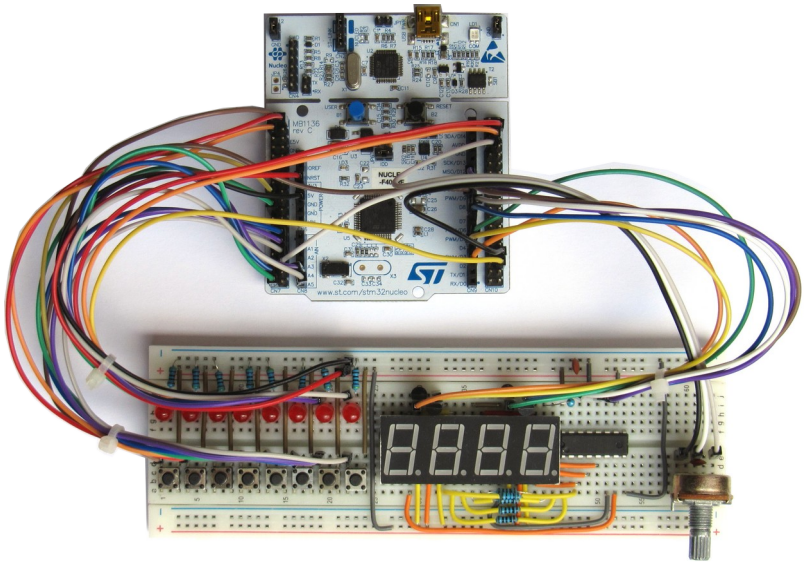
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

Mikroprozessoren-Labor



Versuch 1, GPIO

Versuchsvorbereitung

Stand: 11. November 2021

I. Allgemeine Informationen

Die Informationen in diesem Abschnitt gelten für alle Laborvorbereitungen, Labordurchführungen und Labornachbereitungen.

1 STM32CubeIDE

Die Installation und Verwendung der STM32CubeIDE wird im Moodle-Kurs zur Vorlesung über Video-Tutorials beschrieben. Da die STM32CubeIDE aus Eclipse mit einigen Plugins besteht, wird diese Entwickungsumgebung im folgenden auch einfach als „Eclipse“ bezeichnet, insbesondere wenn auf Eigenschaften oder Funktionen Bezug genommen wird, die nicht von den STM32CubeIDE-spezifischen Erweiterungen bereit gestellt werden, sondern allgemein in Eclipse verfügbar sind.

2 Ausgangsprojekt

Für jedes Labor wird Ihnen im Moodle-Kurs zu dem Labor ein Ausgangsprojekt bereitgestellt. Importieren Sie dieses Projekt für jedes Labor als „Startprojekt“ in Eclipse. Die Ausgangsprojekte enthalten neben dem Mbed OS i. allg. bereits vorbereitete, teilweise oder vollständig implementierte Klassen.

Sie benötigen die Startprojekte bereits für die Vorbereitung des jeweiligen Labors.

3 Programmierrichtlinien

Abgaben, die die nachfolgenden Richtlinien nicht einhalten werden mit einer „gelben Karte“ bewertet (bzw. im Wiederholungsfall mit einer „roten Karte“).

3.1 Allgemeine Regeln

Es gelten folgende Namenskonventionen:

- Namen und Bezeichnungen in Englisch
- Kommentare in Deutsch oder Englisch
- Großbuchstaben zum Abgrenzen von Namensbereichen, sonst Kleinschreibung (z. B. `getName`)
- Bezeichnung von Methoden mit Verben (z. B. `displayProperties()`)
- Bezeichnung von Attributen und Klassen mit Substantiven
- Großschreibung von Konstanten (`#defines` und `enums`)
- Päfixe:
  - `C` für Klassen (z. B. `CPoint`)
  - `m_` für Attribute (z. B. `m_maxvalue`)

- Die Blöcke nach `if`, `else`, `while`, `for`, `do` müssen in geschweiften Klammern eingeschlossen werden, auch wenn Sie nur eine Zeile enthalten.
- Für den Code dürfen beim Übersetzen keine Warnungen ausgegeben werden. Falls Sie eine Warnung nicht korrigieren können, beschreiben Sie in einem Kommentar, was Sie versucht haben bzw. wieso Sie meinen, dass die Warnung nicht berechtigt ist.

Kommentieren Sie alle Klassen und Methoden für die Verarbeitung der Kommentare entsprechend diesem Style Guide: http://mesos.apache.org/documentation/latest/doxygen-style-guide/. Diese Vorgabe soll einen einheitlichen Kommentierungsstil sicherstellen und bereitet Sie auf die Verwendung des Werkzeugs Doxygen in Software Engineering vor. Die Kommentare für Klassen, Methoden und Funktionen müssen als vollständige, grammatikalische korrekte Sätze formuliert sein. Sie sollen die Aufgabe der Klasse bzw. Methode kurz aber vollständig beschreiben. Das Kopieren von Textpassagen aus der Aufgabenstellung als Kommentar ist nicht zulässig, da die Formulierungen in der Aufgabenstellung einem anderen Zweck dienen und als Quell-Code-Kommentare i. allg. nicht geeignet sind.

Wählen Sie für Klassen, Methoden und Variablennamen „selbsterklärende“ Bezeichner. Außer in Ausnahmefällen (z. B. „`i`“ für eine Laufvariable ohne weitere Bedeutung[1]) darf kein Bezeichner weniger als 3 Zeichen haben (typisch sind mehr als sechs Zeichen).

Die Kommentierung einzelner Code-Zeilen ist bei sinnvoll gewählten Bezeichnern normalerweise unnötig.

Zeilen dürfen niemals länger als 80 Zeichen sein. Sie können in Eclipse über „Window | Preferences | General | Editors | Texteditors“ die Anzeige einer Begrenzung als Hilfe für das Einhalten dieser Regel einschalten. Diese Regel gilt auch für Zeilen, die (nur) Kommentare enthalten.

Bei der Methoden-Implementierung gibt es zwischen den Code-Zeilen i. allg. keine Leerzeilen. Wenn Sie das Gefühl haben, Code-Zeilen mit Hilfe von Leerzeile gruppieren zu wollen, entspricht das dem Gefühl in einem „normalen“ Dokument einen neuen Absatz anfangen zu wollen. In diesem Fall können Sie eine Leerzeile einfügen. Die Leerzeile muss dann von einer Kommentarzeile gefolgt sein, in der beschrieben wird, was in der folgenden Gruppe von Code-Zeilen gemacht wird. Gruppen von Code-Zeilen dürfen maximal 5 Zeilen enthalten. (Das bedeutet insbesondere auch, dass es nie mehr als 5 aufeinander folgende Code-Zeilen ohne Unterbrechung durch einen Kommentar gibt.) Da der Beginn eines neuen Blocks optisch bereits eine Abgrenzung darstellt, kann die Leerzeile vor einem Kommentar am Anfang des Blocks (also einem Kommentar unmittelbar nach der öffnenden geschweiften Klammer) entfallen.

3.2 Anmerkungen

Da es nach den Erfahrungen der vergangenen Semester vielen nicht gelingt, die obigen Regeln sinnvoll insbesondere im Zusammenhang mit einem `if`/`else`-Konstrukt anzuwenden, und in diesem

1 „Ohne weitere Bedeutung“ kommt erstaunlich selten vor. Wenn Sie z. B. über die Zeilen und Spalten einer Matrix iterieren, nennen Sie die Laufvariablen natürlich nicht „i“ und „j“ sondern „row“ und „column“.

Zusammenhang auch manchmal die Frage nach mehrzeiligen Kommentaren aufgekommen ist, hier einige Anmerkungen auf der Basis der Erfahrungen aus den vergangenen Semestern.

Antipattern (schlecht):

```
/*
 * Jetzt wird geprüft, ob ein bestimmter Fall eingetreten ist. Wenn der Fall
 * eingetreten ist, sollen folgende Dinge getan werden ... Wenn nicht,
 * folgende Dinge...
 */
if (Bedingung) {
    Anweisung1;
    Anweisung2;
} else {
    Anweisung3;
    Anweisung4;
}
```

Diese Kommentierungsstrategie ist schlecht, weil sie den Kommentar und die zugehörigen Anweisungen räumlich voneinander trennt. Ich habe sie besonders bei Teilnehmern beobachtet, die in der ersten Version gar keine Implementierungskommentare hatten, und statt dessen versucht haben, die Implementierung im Funktionskommentar zu erläutern (was natürlich überhaupt nicht sinnvoll ist, denn den Nutzer einer Funktion interessieren die Implementierungsdetails einer Funktion im Normalfall überhaupt nicht).

Statt die Kommentierung grundsätzlich zu überdenken, haben viele Betroffene dann versucht, möglichst viel vom ursprünglichen Kommentar "zu retten" und Teile davon als mehrzeiligen Kommentar in die Implementierung kopiert.

Die bessere Form der Kommentierung für das obige Beispiel wäre:

```
// Ist Fall ... eingetreten?
if (Bedingung) {
    // ... ist eingetreten, daher folgendes Dinge tun ...
    Anweisung1;
    Anweisung2;
} else {
    // Da ... nicht eingetreten ist, alternativ ...
    Anweisung3;
    Anweisung4;
}
```

(Bitte nicht meine abstrakten Formulierungen als Muster übernehmen! Im konkreten Fall gibt es sehr viel bessere.)

Wie man sieht, ist jetzt der mehrzeilige Kommentar aus dem schlechten Beispiel zu drei, im Normalfall einzeiligen Kommentaren geworden, die unmittelbar vor dem Code stehen, den sie kommentieren. Da kein vernünftiger Mensch auf die Idee kommt, einzeilige Kommentare anders als mit dem "`//`" am Anfang zu schreiben, sollte sich damit auch die (manchmal aufgeworfene) Frage nach

mehrzeiligen Kommentaren normaler Weise gar nicht stellen. (Nach den zwei Schrägstrichen kommt übrigens immer ein Leerzeichen, ebenso wie nach dem Stern eines mehrzeiligen Kommentars, das gebietet schon die Ästhetik).

Mehrzeilige Implementierungskommentare sind nicht generell „verboten“, sie sind aber ein eindeutiger Hinweis darauf, dass bei der Kommentierung etwas grundsätzlich falsch gemacht wurde. Wenn eine Gruppe von maximal fünf Code-Zeilen mehr als ein oder vielleicht zwei Kommentarzeilen benötigt, haben Sie entweder keine sprechenden Bezeichner verwendet oder es handelt sich um wirklich geniale Code-Zeilen. (Auch wenn ein Implementierungskommentar mal zweizeilig (oder auch dreizeilig) wird, sollten Sie ihn übrigens immer besser mit "`//`" am Anfang der Zeilen schreiben und nicht mit "`/* ... */`", das ist als Implementierungskommentar wesentlich besser lesbar.)

4 Aufgabenbearbeitung

Alle Aufgaben aus der Laborvorbereitung und der Labordurchführung, die nicht explizit als „freiwillig“ gekennzeichnet sind, müssen bearbeitet werden.

5 Labordokumentation

Zusätzlich zu der ZIP-Datei mit dem aus Eclipse exportierten Code geben Sie nach jedem Labor eine Labordokumentation als PDF ab. Die Inhalte der Labordokumentation ergeben sich für jedes Labor aus den in der Versuchsvorbereitung oder Versuchsdurchführung mit „✎“ gekennzeichneten Abschnitten. Kopieren Sie die mit „V-n“ oder „D-n“ gekennzeichneten Abschnitte in Ihre Labordokumentation und schreiben Sie die Antwort danach in einen neuen Absatz (bzw. bei Bedarf in mehrere Absätze).

II. Versuchsbeschreibung

In diesem Versuch sollen die GPIO-Pins des Prozessors verwendet werden, um Tasten einzulesen und LEDs anzusteuern.

III. Aufgaben

1 Installation und Aufbau

Installieren Sie die STM32CubeIDE entsprechend der Anleitung im Moodle-Vorlesungskurs.

Bauen Sie die Schaltung auf dem Bread-Board auf (Taster und LEDs) und nehmen Sie sie entsprechend der Anleitung in Betrieb.

2 Modellieren der LEDs

Legen Sie ein globales Objekt `leds` vom Typ `BusOut` an, mit dem Sie die LEDs ansteuern können. Implementieren und testen Sie die Funktion `task1()` (s. Ausgangsprojekt im Moodle-Kurs). Die Funktion soll alle 8 LEDs alle 100 ms ein- und wieder ausschaltet.

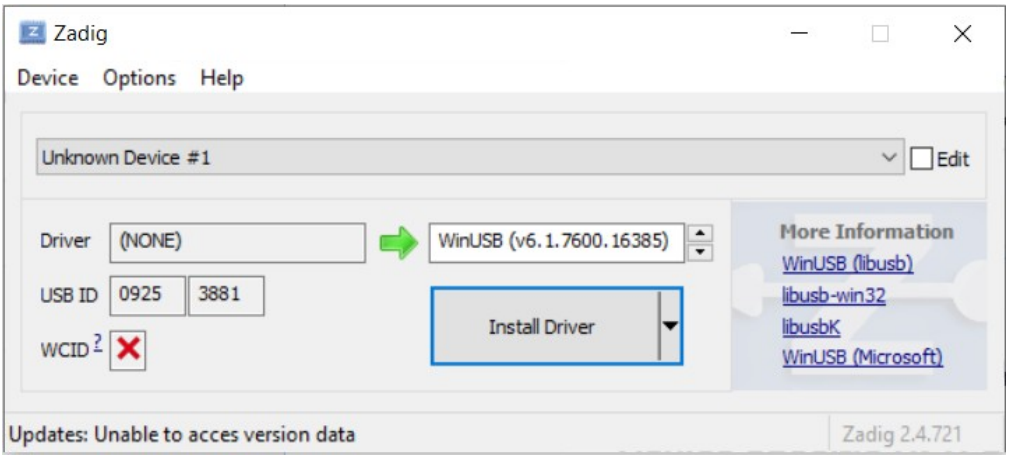
3 Inbetriebnahme Logic-Analyzer

3.1 Installation der Software

Für die Nutzung des Logik Analysators nutzen Sie im Labor das Programm PulseView. Laden Sie es von der Download-Seite (https://sigrok.org/wiki/Downloads) herunter und installieren Sie es.

3.2 Konfiguration des Device-Treiber

Schließen Sie den Logik-Analysator an den PC bzw. Ihr Notebook an. Er wird von Windows als „Unknown Device“ angelegt, da Windows keinen geeigneten Treiber beinhaltet. Klicken Sie auf das Windows-Logo und tippen Sie „Zadig“ ein, um die Applikation für die Treiberinstallation zu starten (Details s. https://sigrok.org/doc/pulseview/0.4.1/manual.html#windows). In dem daraufhin erscheinenden Dialog klicken Sie auf „Install Driver“, um für das „Unknown Device“ den Treiber „(None)“ durch den „WinUSB“-Treiber zu ersetzen.



3.3 Erste Analyse

Ein Logik-Analysator zeichnet die logischen Signale (Low/High bzw. 0/1) an verschiedenen Eingängen auf. Wenn Sie so wollen, ist ein Logik-Analysator ein „Vielkanal“ (in unserem Fall 8-Kanal) Speicheroszilloskop. Da sich die Signalerfassung für jeden Kanal aber auf die Unterscheidung zwischen Low- und High-Pegel beschränkt, kann ein Logik-Analysator wesentlich günstiger realisiert werden.

Die Bedienung ist im Manual beschrieben (https://sigrok.org/doc/pulseview/0.4.1/manual.html#_data_acquisition). Als Treiber für den AZ-Delivery Logik-Analysator wählen Sie „fx2lafw“.

Schließen sie GND und die Eingangskanäle des Logik-Anaylsators an das Evaluations-Board an. Dafür können Sie die Pfostenstecker auf der Rückseite des Evaluations-Boards verwenden.

Stellen Sie die niedrigste Abtastrate (20 kHz) ein, da die LEDs relativ langsam blinken (Periode 5 Hz). Wählen Sie Anzahl der Stichproben (samples) so, dass Sie 5 Perioden aufzeichnen.

4 Modellieren der Taster

Legen Sie ein globales Objekt `keys` vom Typ `BusIn` an, mit dem Sie den Zustand der Taster einlesen können. Da es nicht möglich ist, schon beim Aufruf des Konstruktors die Pull-Down Widerstände einzuschalten, erledigen Sie das, indem Sie als erste Anweisung in der `main`-Funktion die Methode `mode` mit dem entsprechenden Argument aufrufen.

Implementieren und testen Sie die Funktion `task2()`. Die Funktion soll bewirken, dass eine LED leuchtet, solange der Taster unterhalb der LED gedrückt ist (die Funktion besteht aus einem „Einzeiler“ in einer Endlosschleife).