



Implementierung einer Ampelsteuerung

Klausurersatzleistung Embedded Systems in der Automation

im Studiengang Elektrotechnik

an der Dualen Hochschule Baden-Württemberg Mosbach

von

Timo Kempf

20.06.2022

Matrikelnummer

5183010

Ausbildungsfirma

Wirl Elektrotechnik GmbH

Gruppenmitglieder

Cedric Franke, Sven Mößner, Niklas Stein

Dozent:in

Claudia Heß

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Klausurersatzleistung mit dem Thema: *Implementierung einer Ampelsteuerung* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1 Einleitung	1
1.1 Aufgabenstellung	1
1.2 Vorgehensweise	1
2 Hardware	3
2.1 Allgemeines	3
2.2 Aufbau	3
2.3 Pinbelegung	4
3 Systemarchitektur	5
3.1 Klassendiagramm des Projekts	5
3.2 Zustandsautomat	8
4 State Pattern	9
4.1 Context, Interface und Concrete Klassen	9
4.2 Singleton	9
5 Hardwarezugriff - GPIO	12
6 Wechsel zwischen Hardware- und Softwarebetrieb	16
6.1 Definition	16
6.2 Umsetzung	16
6.2.1 Ausgabeformat	17
6.2.2 Eingabeformat	18
7 Fazit	20
Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
Quellcodeverzeichnis	V

1 Einleitung

In der Vorlesung „Embedded Systeme in der Automation“ geht es darum zu lernen, wie ein Mikrocontroller objektorientiert, effektiv und mit möglichst wenig Speicheranforderung programmiert werden kann. Dazu wurden in der Vorlesung einige nützliche Methoden übermittelt. Gegen Ende des Semesters wird in diesem Modul jedoch keine Klausur geschrieben. Stattdessen bekommen die Studierenden ein Projekt zur praktischen Anwendung sämtlicher Methoden. Das Ziel dieser Projektarbeit ist es eine Ampelsteuerung zu implementieren.

1.1 Aufgabenstellung

Die in diesem Projekt zu erstellende Ampel soll, wie auch die Ampeln im Straßenverkehr, zwei Betriebszustände haben. Diese sind „active“ und „flashing“. Ersterer Zustand soll eine normale Ampelsteuerung ermöglichen mit den Farbwechseln Rot → Rotgelb → Grün → Gelb → Rot. Der zweite Zustand soll signalisieren, dass die Ampel nicht in Betrieb ist. Dies soll dadurch erreicht werden, dass das gelbe Licht blinken soll. Weiterhin soll es für den Benutzer möglich sein die Schnittstellen zur Ampelsteuerung selbst zu wählen. Hier soll es zwei Möglichkeiten geben. Erstere soll die Hardwareschnittstelle mit zwei Buttons und drei LEDs mit den Farben rot, gelb und grün sein. Andernfalls soll es auch möglich sein die Ampel über das Terminal zu steuern. Dies wäre die Softwareschnittstelle.

1.2 Vorgehensweise

Die Vorgehensweise bei diesem Projekt orientiert sich an einem Leitfaden, welcher von der Dozentin Claudia Heß stammt. Zunächst werden die bereitgestellte Hardware analysiert und erste Vereinbarungen bezüglich der Pinbelegungen getroffen. Danach wird ein Styleguide angelegt. Dieser ist nötig um einen möglichst einheitlichen Code zu erhalten. Im nächsten Schritt wird dann eine Architektur entworfen

um alle geforderten Funktionen zu erfüllen. Daraufhin werden verwendete State Patterns vorgestellt und erklärt. Zuletzt wird dann noch auf den Hardwarezugriff eingegangen, bevor gezeigt wird, wie der Wechsel zwischen Hardware- und Softwareschnittstelle erfolgt. Zum Schluss wird dann ein Fazit gezogen.

2 Hardware

Dieses Kapitel befasst sich mit der im Projekt eingesetzten Hardware.

2.1 Allgemeines

Verwendet wird das STM32F401RE Nucleo Board der Firma STMicroelectronics. Weiterhin wird ein Shield verwendet, welches auf den Microcontroller aufgesteckt werden kann. So wird eine Verbindung zwischen einigen GPIOs des Boards und dem Shield ermöglicht. Des Weiteren werden LEDs und Buttons mitgeliefert. Zuletzt wird noch ein USB-Kabel zur Verbindung zwischen Laptop und Mikrocontroller benötigt. Dieses versorgt den Controller mit Spannung und ermöglicht eine serielle Kommunikation, welche später benötigt wird.

2.2 Aufbau

Im Folgenden wird nun auf den Aufbau der Hardware eingegangen. Das Shield wird auf das Board aufgesteckt. Die mitgelieferten LEDs und Buttons können nun einfach am Shield eingesteckt werden und erhalten damit eine leitende Verbindung zu Versorgungsspannung, Masse und einer Signalleitung. Die Verbindung über das Shield bietet eine höhere Festigkeit und Stabilität gegenüber einzelnen Jumperverbindungen, welche direkt am Board gemacht werden könnten.

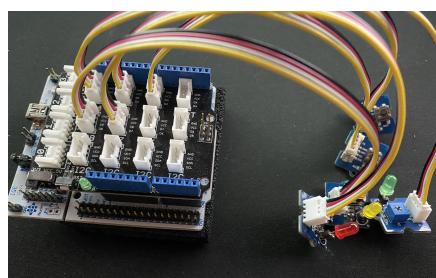


Abbildung 2.1: Hardwareaufbau der Ampelsteuerung

Quelle: Eigene Darstellung

2.3 Pinbelegung

Da der Arbeitsaufwand dieses Projektes auf vier Studierende aufgeteilt wird, muss abgestimmt werden, welcher Pin jeweils den LEDs und den Buttons zugeordnet werden soll. Deshalb wird gleich zu Beginn des Projektes eine Vereinbarung der Pinbelegungen getroffen. Diese ist in Tabelle 2.1 zu sehen.

Hardware - Shield	Software (Port Pin)	Bauteil
D7	A8	LED Rot
D8	A9	LED Gelb
D2	A10	LED Grün
D3	B3	Taster 1
D4	B5	Taster 2

Tabelle 2.1: Pinbelegung (Hardware/Software)

3 Systemarchitektur

3.1 Klassendiagramm des Projekts

In diesem Abschnitt wird das Klassendiagramm der Ampelschaltung (siehe Abbildung 3.1) erläutert. Es wird darauf eingegangen, wie die einzelnen Klassen in Beziehung zueinanderstehen und wie diese gruppiert werden können.

Das Klassendiagramm lässt sich in vier Bereiche unterteilen:

- Auswahl Betriebsmodus (blau)
- Steuerung Farben (grün)
- Output (rot)
- Input (schwarz)

Der Bereich „Auswahl Betriebsmodus“ bildet die Logik der beiden Betriebsmodi ab, in denen sich die Ampel befinden kann. Zum einen der aktive Modus, in welchem die Ampel die Farben durchschaltet und zum anderen den Modus, in welchem die Ampel orange blinkt. Der Bereich „Steuerung Farben“ übernimmt die Aufgabe die Zustände im Hinblick auf die Farben der Ampel bereitzustellen. Die Ampel kann Rot ausgeben, Rot-Organge, Orange, Grün oder ausgeschaltet sein. Der Output übernimmt die Funktion, die einzelnen Farben zu setzen und dies entweder software- oder hardwaremäßig. Der Input übernimmt die Aufgabe die Benutzereingabe bereitzustellen, die ebenso entweder software- oder hardwaremäßig eingegeben werden kann. Die Klasse GPIO, die nicht in einen der Bereiche zugeordnet wird, bildet die Schnittstelle für den Hardwarezugriff ab.

Hierarchisch ganz oben ist die Kontextklasse „Traffic Light“ zu sehen. Ein Objekt dieser Klasse wird in der `main()` erstellt und beinhaltet die Logik der gesamten

Ampelschaltung. Wird die `Handle()` Funktion dieses Objekts aufgerufen, läuft die Ampelschaltung wie gefordert ab. Die Aufgabe Benutzereingaben einzulesen wird an das Interface „InputFormat“ delegiert. Je nachdem, ob eine Hardware angeschlossen ist, liefern die Klassen „SoftwareInput“ und „HardwareInput“ den jeweiligen Tasterwert, bzw. die Tastatureingabe. Die Klasse „UserButtons“ frägt beide Pins (für die beiden Taster) ab und stellt der Klasse „HardwareInput“ nur einen Wert zur Verfügung, der die beiden Tasterzustände abbildet. Damit die Klasse „UserButtons“ auf die Registerinhalte des Mikrocomputers zugreifen kann, steht sie in Beziehung mit der Klasse „GPIO“. Ein Objekt der „Traffic Light“ Klasse kann den Zustand in Betrieb oder außer Betrieb annehmen. Über das Interface „state“ werden die Zustandsklassen „flashing“ und „active“ hierfür bereitgestellt. Diese beiden Klassen greifen über das Interface „LightControl“ auf die Farbenklassen „Off“, „Red“, „RedAmber“, „Amber“ und „Green“ zu. Diese Unterklassen sind dafür verantwortlich, dass die Ampel den Farbzustand annimmt. Des Weiteren bieten diese Klassen die Funktion an, Objekte für den nächsten Farbzustand zurückzuliefern. Noch eine Hierarchieebene weiter darunter greifen die Farbklassen über das Interface „OutputFormat“ auf die „SoftwareOutput“ bzw. auf die Klasse „HardwareOutput“ zu. Diese realisieren die Ausgabe der Ampelschaltung. Im Fall, dass Hardware angeschlossen ist, greift die „HardwareOutput“ Klasse über die „UserLEDs“ Klasse auf die „GPIO“ Klasse zu. Damit werden die entsprechenden Bits in den Registern des Mikrocontrollers gesetzt, damit die entsprechenden Pins der LEDs auf HIGH geschaltet werden können.

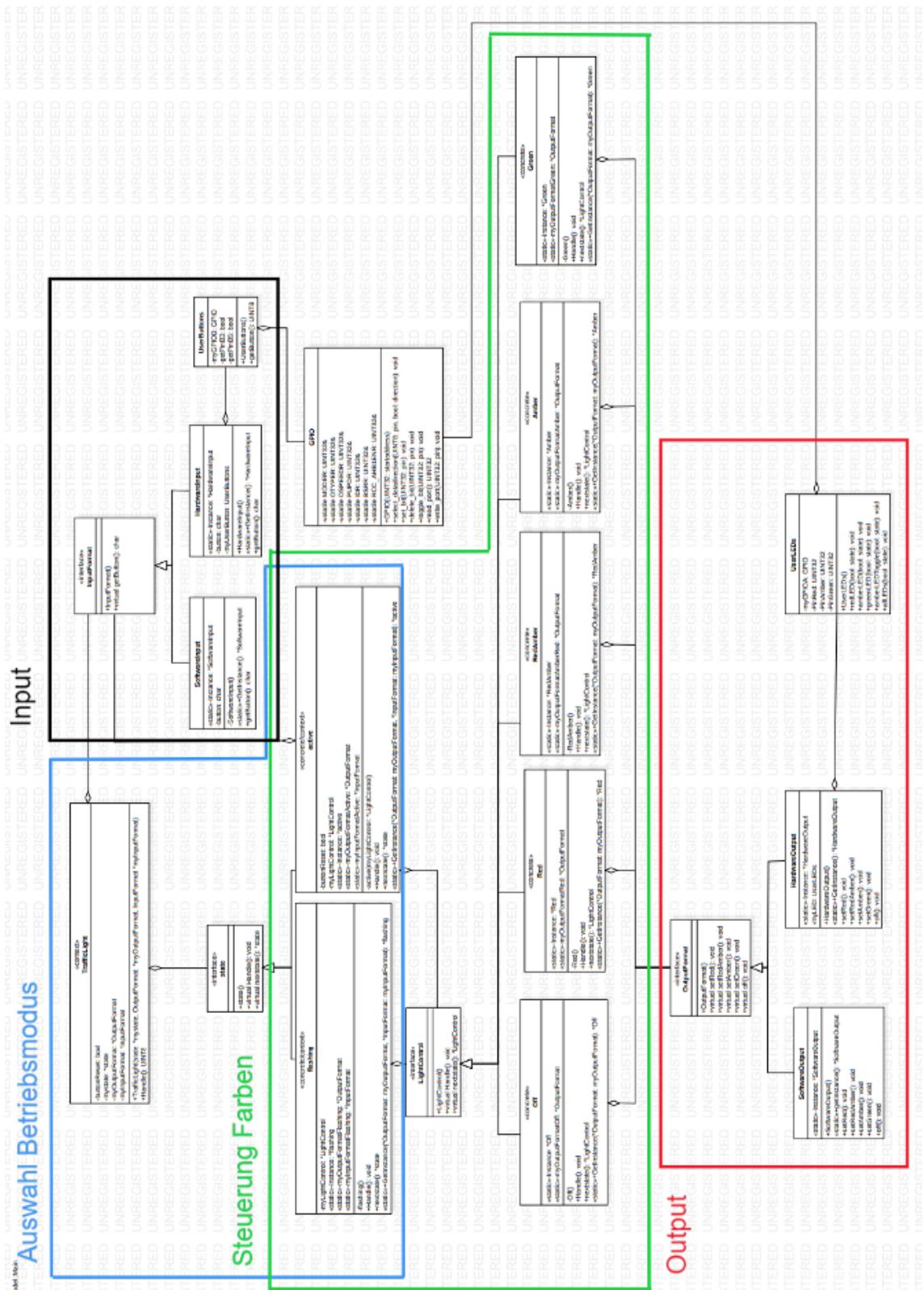


Abbildung 3.1: Klassendiagramm der Ampelsteuerung
Quelle: Eigene Darstellung

Timo Kempf (20.06.2022)

3.2 Zustandsautomat

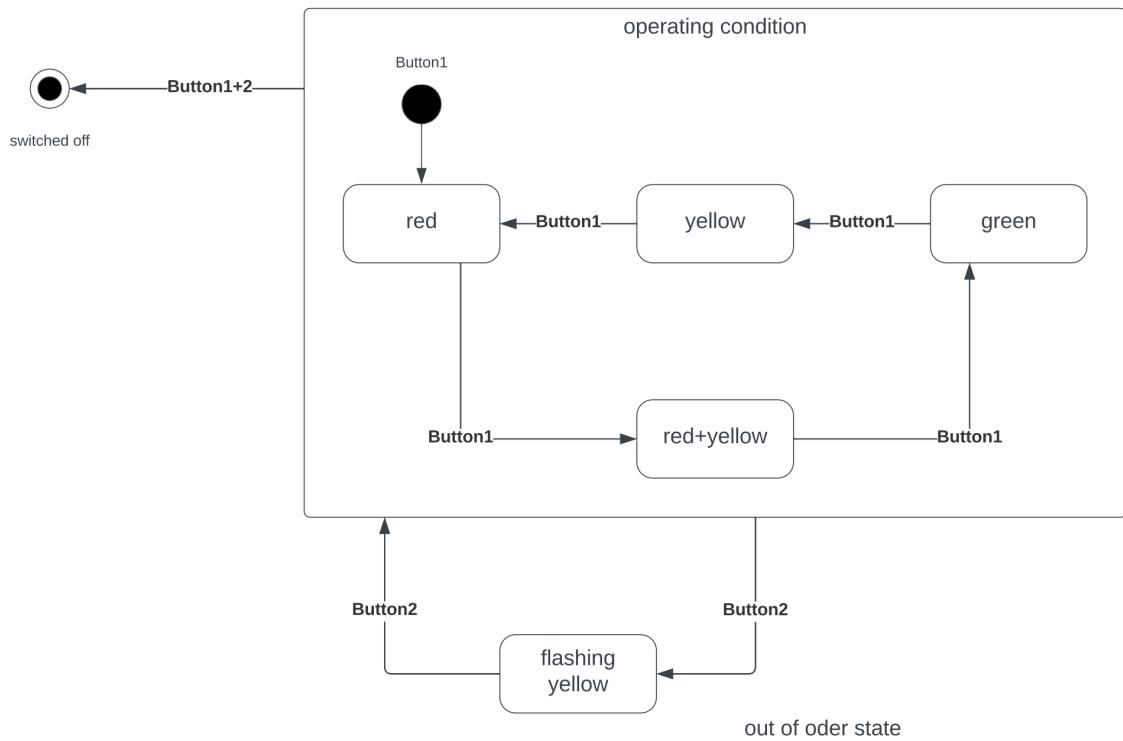


Abbildung 3.2: Zustandsautomat der Ampelsteuerung
 Quelle: Eigene Darstellung

Ein Zustandsdiagramm ist eines der 14 Diagrammarten der Sprache UML für Software und andere Systeme. Es ermöglicht dem Entwickler sich einen Gesamtüberblick über die Zustände und Operationen, welche in einem Projekt/Programm erfolgen sollen, zu verschaffen.

Die zu implementierende Ampelsteuerung soll zwei Zustände „In Betrieb“ und „Außer Betrieb“ annehmen können. Im Betrieb soll die Ampelsteuerung die Ampelphasen Rot, Rot-Gelb, Grün und Gelb durchlaufen (siehe Abbildung 3.2). Die Ampelphasen sollen per Taster „T1“ oder Tastatureingabe „F“ gesteuert werden. Zwischen jeder Ampelphase kann die Ampelsteuerung mit dem Taster „T2“ oder Tastatureingabe „B“ in den „Außer Betrieb“ Zustand gesetzt werden. Innerhalb dieses Zustandes, soll die gelbe LED blinken (siehe Abbildung 3.2). Immer wenn die Ampelsteuerung neugestartet oder in den Betriebsmodus „In Betrieb“ wechselt, soll sie in der roten Ampelphase starten.

4 State Pattern

Nachfolgend wird das eingesetzte Design Pattern State erläutert.

4.1 Context, Interface und Concrete Klassen

In der vorliegenden Ampelschaltung werden zwei State Patterns verwendet. Zudem wird beim Input und Output sich eines Interfaces bedient, welches Polymorphie zulässt. Das heißt, je nachdem ob Hardware angeschlossen ist, werden die Hardware oder Software Klassen verwendet. Näheres dazu wird in dem Kapitel „Wechsel zwischen Hardware- und Softwarebetrieb“ beschrieben. Das hierarchisch höhere State Pattern besteht aus der Kontextklasse „TrafficLight“, dem Interface „state“ und den Concrete Klassen „flashing“ und „active“. Je nach Benutzereingabe verweist der Zeiger vom Typ „state“ auf eine Instanz der Klasse „flashing“ oder „active“. Wird die Handle Funktion des Interfaces aufgerufen, ist die Ampel entweder im konkreten Zustand „in Betrieb“ oder „außer Betrieb“. Fordert der Benutzer ein Wechsel des Betriebsmodus, so ist der nächste Zustand der jeweils andere Betriebsmodus. Das zweite State Pattern wechselt zwischen den Farbzuständen. Dieses State Pattern besteht entweder aus der Kontextklasse „flashing“ oder „active“, dem Interface „LightControl“ und den Concrete Klassen („Off“, „Red“, „RedAmber“, „Amber“, „Green“). Die Klasse „active“ ruft die Funktionen „nextstate“ des Interfaces auf, damit die Ampelfarben durchgeschaltet werden. Im außer Betrieb Zustand wechselt der Farbzustand der Ampel zwischen Off und Amber.

4.2 Singleton

Das Singleton Design Pattern hilft in der objektorientierten Programmierung dem Entwickler mit vielfachen Vorlagen das Lösen von Programmieraufgaben. Es ist sehr leistungsfähig und gehört zu der Kategorie der Erzeugungsmuster unter den Design Pattern. Der Ausdruck Singleton wird oftmals als „Einzelstück“ betitelt. Dieses Synonym bildet sehr gut die Funktion einens Singeltons ab, da die Aufgabe

darin besteht nicht mehr als ein Objekt einer Klasse zu erstellen. Wurde mit der Klasse über das Singleton Pattern nur eine Instanz der Klasse erzeugt, sorgt es auch dafür, dass es lediglich bei dieser einen Instanz der Klasse bleibt. Oftmals dient eine Funktion, wie `GetInstance` dafür, dass nur eine Instance der Klasse erstellt wird.

Im Bezug auf die Ampelsteuerung wurde das Singleton Pattern in sieben unterschiedlichen Klassen verwendet. Diese lauten wie folgt:

- active.h
- flashing.h
- Red.h
- Amber.h
- RedAmber.h
- Green.h
- Off.h

Da der Aufbau in jeder Klasse identisch ist, folgt die Erläuterung im Code nun am Beispiel der Klasse `Red.h`.

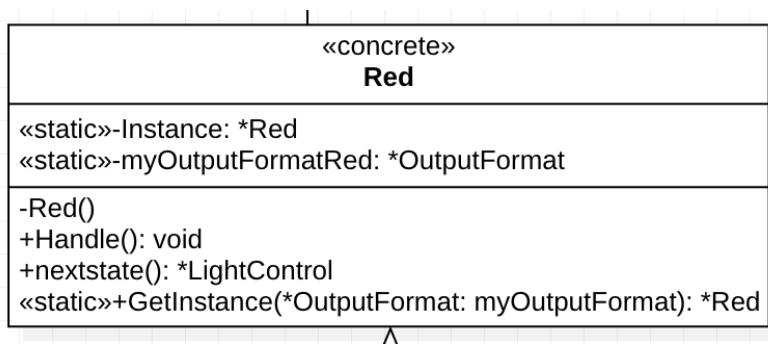


Abbildung 4.1: Singleton Patten am Beispiel der Red.h Klasse
 Quelle: Eigene Darstellung

In der Abbildung 4.1 sind die Funktionen und Instanzen der `Red.h` Klasse zu sehen, welche mit Hilfe des Singleton Pattern implementiert wurde. Das Schlüsselwort

`static` bedeutet einmal, dass die als `static` deklarierte Variable oder Funktion ihren Wert innerhalb zwischen Aufrufen behält und andermal, dass eine globale statische Variable oder Funktion nur in der Datei „gesehen“ wird, in der sie deklariert wurde. Um das Singleton Pattern nutzen zu können, wird die statische Instanz und die Funktion `GetInstance` benötigt. Die Funktion `GetInstance` hat die Aufgabe, bei jedem Aufruf eine Instance zu erzeugen (siehe Codeauszug 4.1).

```
1      ...
2      Red *Red::Instance = NULL;
3
4      Red *Red::GetInstance(OutputFormat *myOutputFormat)
5      {
6          myOutputFormatRed = myOutputFormat;
7          if (Instance == NULL)
8          {
9              Instance = new Red();
10         }
11         return Instance;
12     }
13     ...
```

Codeauszug 4.1: Red.cpp - Singleton in der Klasse Red

Wie in Zeile 2 zu erkennen ist, wird die Instanz der Klasse immer auf `NULL` deklariert. In Zeile 4 der cpp-Datei wird ist die Implementierung der Funktion `GetInstance` zu sehen. Innerhalb dieser Funktion wird der übergebene Parameter `myOutputFormat` der Klasse Red zugewiesen. Folgend findet eine `if` Überprüfung statt, ob es schon eine Instance der Klasse Red.h gibt. Wenn nicht, wird in Zeile 9 eine neue Instanz der Klasse Red.h erzeugt. Anschließend besitzt die Funktion `GetInstance` als return-Wert die erzeugte Instanz. Wichtig ist es, dass die statische Instanz als `private` und die Funktion `GetInstance` als `public`.

5 Hardwarezugriff - GPIO

Der Hardwarezugriff erfolgt über drei Klassen. Die untergeordnete Klasse `GPIO` greift dabei direkt auf die Register des Mikrocontrollers zu und steuert somit sämtliche Hardwarezugriffe. Die zwei übergeordneten Klassen `UserLEDs` und `UserButtons` greifen dann auf die Klasse `GPIO` zu und steuern diese. Übergeordnet heißt hier jedoch nicht, dass es sich um Oberklassen einer Vererbung handelt. Die hier herrschende Beziehung ist eine Aggregation. Außer zu diesen beiden Klassen hat die `GPIO`-Klasse keine weiteren Verbindungen. Eine Hardwareansteuerung kann somit nur durch die beiden genannten übergeordneten Klassen durchgeführt werden.

Nun wird etwas genauer auf die die `GPIO`-Klasse eingegangen. Sie ist wie folgt aufgebaut:

GPIO
<code>-volatile MODER: UINT32&</code> <code>-volatile OTYPER: UINT32&</code> <code>-volatile OSPEEDR: UINT32&</code> <code>-volatile PUPDR: UINT32&</code> <code>-volatile IDR: UINT32&</code> <code>-volatile BSRR: UINT32&</code> <code>-volatile RCC_AHB1ENR: UINT32&</code>
<code>+GPIO(UINT32: startaddress)</code> <code>+select_datadirection(UINT8: pin, bool: direction): void</code> <code>+set_bit(UINT32: pin): void</code> <code>+delete_bit(UINT32: pin): void</code> <code>+toggle_bit(UINT32: pin): void</code> <code>+read_port(): UINT32</code> <code>+write_port(UINT32: pin): void</code>

Abbildung 5.1: GPIO Klasse
 Quelle: Eigene Darstellung

Alle Attribute dieser Klasse sind als „private“ gekennzeichnet. Das liegt daran, dass es sich bei den Attributen um Zeiger handelt, mit welchen man auf die `GPIO`-Register des Mikrocontrollers zugreifen kann. Hier wird das Prinzip des Information Hidings angewendet, um zu verhindern, dass die Register von außen manipuliert werden könnten. Weiterhin wird den Attributen das Schlüsselwort `volatile` vorangestellt. Dieses gibt dem Compiler die Information, dass sich der Inhalt auf den die

Zeiger zeigen, ändern kann, auch wenn das Programm nicht schreibend darauf zugreift.

Der Konstruktor der Klasse konfiguriert einen GPIO-Port. Ihm muss lediglich die Startadresse des gewünschten Ports übergeben werden. Diese Startadresse kann dem Datenblatt auf Seite 53 entnommen werden. Auf Basis der Startadresse adressiert dieser dann die benötigten Register. Weiterhin wird dort das Clock Signal des ausgewählten Ports aktiviert. Über die Methode `select_datadirection()` kann dann für jeden Pin des Ports einzeln ausgewählt werden ob, dieser als Input oder als Output fungieren soll. Dazu muss der Methode nur die Pinnummer und die Art des GPIOs übergeben werden. Mit `set_bit()` kann ein ausgewählter Output gesetzt werden. Dazu muss lediglich die Pinnummer bekannt sein. `delete_bit()` macht genau das Gegenteil. Dort wird ein Output zurückgesetzt. `toggle_bit()` vereint beide Methoden. Dort wird der Status des Outputs abgefragt und danach umgekehrt. Mit `read_port()` werden die Zustände jedes einzelnen Pins des gesamten Ports gelesen und als 32-Bit Wert zurückgegeben. Bei der Ausführung von `write_port()` hingegen, wird der gesamte Port beschrieben. Wirksam ist der Schreibzugriff jedoch nur auf definierte Outputs.

Nun wird auf die Klasse „UserLEDs“ betrachtet. Das dazugehörige Klassendiagramm kann folgender Abbildung entnommen werden:

UserLEDs
-myGPIOA: GPIO -PinRed: UINT32 -PinAmber: UINT32 -PinGreen: UINT32
+UserLEDs() +redLED(bool: state): void +amberLED(bool: state): void +greenLED(bool: state): void +amberLEDToggle(bool: state): void +allLEDs(bool: state): void

Abbildung 5.2: UserLEDs Klasse
 Quelle: Eigene Darstellung

Auch hier sind alle Attribute als private definiert, um das Information Hiding zu

waren. Das erste Attribut ist eine Instanz der `GPIO`-Klasse. Es dient als Schnittstelle zur Ansteuerung der Pins, an welchen die LEDs angeschlossen sind. In den restlichen drei Attributen werden die jeweiligen Pinnummern der LEDs hinterlegt. Dies erfolgt im Konstruktor über eine Initialisierungsliste. Danach wird dort noch, über die `select_datadirection()`-Methode der `GPIO`-Klasse festgelegt, dass es sich bei den vorliegenden Pins um Outputs handelt. Mit den restlichen Methoden werden dann die LEDs angesteuert. Ihnen muss lediglich der gewünschte Zustand übergeben werden.

Zuletzt wird die Klasse `UserButtons` betrachtet. Auch hier wird ein UML Klassendiagramm zur Erläuterung herbeigezogen:

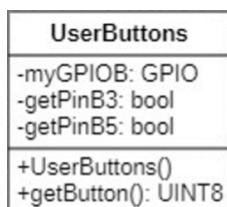


Abbildung 5.3: UserButtons Klasse
 Quelle: Eigene Darstellung

Dort wird auch wieder eine private Instanz der Klasse `GPIO` erstellt. Damit können die Buttons abgefragt werden. Weiterhin existieren zwei private Methoden, welche den Status des jeweiligen Buttons zurückgeben. Sie werden in der Methode `getButton()` aufgerufen. Dort erfolgt eine Entprellung. Dazu wird der Wert zunächst ein erstes Mal abgefragt. Danach wird 3000 Schleifendurchgänge gewartet und der Status erneut abgefragt. Liefert beide Abfragen das gleiche Ergebnis, so wird der Wert übernommen. Andernfalls wird die Entprellung erneut durchgeführt. Wie im Klassendiagramm zu sehen hat die Methode einen Rückgabewert vom Datentyp `UINT8`, dies entspricht einem 8 Bit Charakter. Je nachdem, welcher Button nun gedrückt wird, wird ein bestimmter Rückgabewert zurückgegeben. Eine Auflistung aller möglichen Rückgabewerte kann folgender Tabelle 5.1 entnommen werden:

Welche Buttons gedrückt?	Rückgabewert
Keiner	O
Nur Button an B3 (D3)	B
Nur Button an B5 (D4)	F
Beide Buttons	X

Tabelle 5.1: Rückgabewerte Buttons

6 Wechsel zwischen Hardware- und Softwarebetrieb

Nachstehend wird die Umsetzung der Wechsels zwischen Hardware- und Softwarebetrieb erläutert.

6.1 Definition

Wie bereits erwähnt, soll es dem Anwender möglich sein, die Ampelsteuerung sowohl mit Hardware, als auch komplett ohne Hardware nutzen zu können. Um die beiden Fälle näher zu erläutern, wird der Hardware- und Softwarebetrieb im Folgenden konkret spezifiziert.

- **Hardwarebetrieb:** Der Programmcode der Ampelsteuerung wird auf das Nucleo Board geflasht und auf diesem ausgeführt. Die Nutzereingabe erfolgt über zwei an das Board angeschlossenen Taster. Die Ausgabe der Ampelfarben erfolgt über die drei angeschlossenen LEDs.
- **Softwarebetrieb:** Das Programm der Ampelsteuerung wird auf dem PC ausgeführt. Die Nutzereingabe erfolgt über die Tastatur. Die Tasten „F“ und „B“ repräsentieren die beiden Hardwaretaster. Die Taste „X“ steht für das Betätigen beider Taster gleichzeitig. Das Einlesen der Tastatureingaben erfolgt über den Eingabestream `cin`, die Ausgabe der Ampelfarben über den Ausgabestream `cout` in der Kommandozeile.

6.2 Umsetzung

Ob die Ampelsteuerung im Hardware- oder Softwarebetrieb arbeitet, wird über Pointer bestimmt, welche in der `main.cpp` des Programmes angelegt und bis zur relevanten Stelle im Programmcode durch alle Klassen übergeben werden. Ob die Pointer für den Hardware- oder Softwarebetrieb initialisiert werden, wird wiederum über

eine Präprozessoranweisung festgelegt, welche der Nutzer setzen oder auskommen kann. Dies ist in Codeauszug 6.1 dargestellt.

```
1 ...  
2 #define _HARDWAREPRESENT  
3 int main(){  
4     #ifdef _HARDWAREPRESENT  
5         OutputFormat *myOutputFormat =  
6             HardwareOutput::GetInstance();  
7         InputFormat *myInputFormat =  
8             HardwareInput::GetInstance();  
9     #else  
10        OutputFormat *myOutputFormat =  
11            SoftwareOutput::GetInstance();  
12        InputFormat *myInputFormat =  
13            SoftwareInput::GetInstance();  
14    #endif  
15 ...
```

Codeauszug 6.1: main.cpp - Hardwarebetrieb oder Softwarebetrieb

Im weiteren Programmcode wurde die Umschaltung zwischen Hardware- und Softwarebetrieb nochmals aufgeteilt. Zum einen wird beim Ausgabeformat der Ampelfarben zwischen Hardware- und Softwareausgabe, zum anderen beim Eingabeformat der Nutzereingabe zwischen Hardware- und Softwareeingabe unterschieden.

6.2.1 Ausgabeformat

Abbildung 6.1 zeigt den Ausschnitt aus dem Klassendiagramm, welcher für die Umschaltung zwischen Hardware- und Softwareausgabe zuständig ist. Die Klasse „OutputFormat“ ist eine Interface Klasse und beinhaltet ausschließlich virtuelle Methoden. Die Unterklassen „SoftwareOutput“ und „HardwareOutput“ erben von der Oberklasse „OutputFormat“ und verwenden deren Interface. Die Concrete Klassen des übergelagerten State Patterns, welche die unterschiedlichen Ampelfarben repräsentieren, besitzen den erläuterten übergebenen Zeiger, welcher in der `main.cpp` initialisiert wurde und auf eine Instanz der Klasse „HardwareOutput“ oder „Soft-

wareOutput“ zeigt. Je nachdem werden die Methoden von „HardwareOutput“ oder „SoftwareOutput“ aufgerufen, um die Ampelfarben hardware- oder softwareseitig anzuzeigen.

Beim Hardwarebetrieb ruft die Klasse „HardwareOutput“ schließlich die Methoden der Klasse „UserLEDs“ auf und steuert so die LEDs an. Befindet sich die Ampelsteuerung im Softwarebetrieb, so sorgen die Methoden in der Klasse „SoftwareOutput“ für die entsprechenden Ausgabestreams in der Kommandozeile.

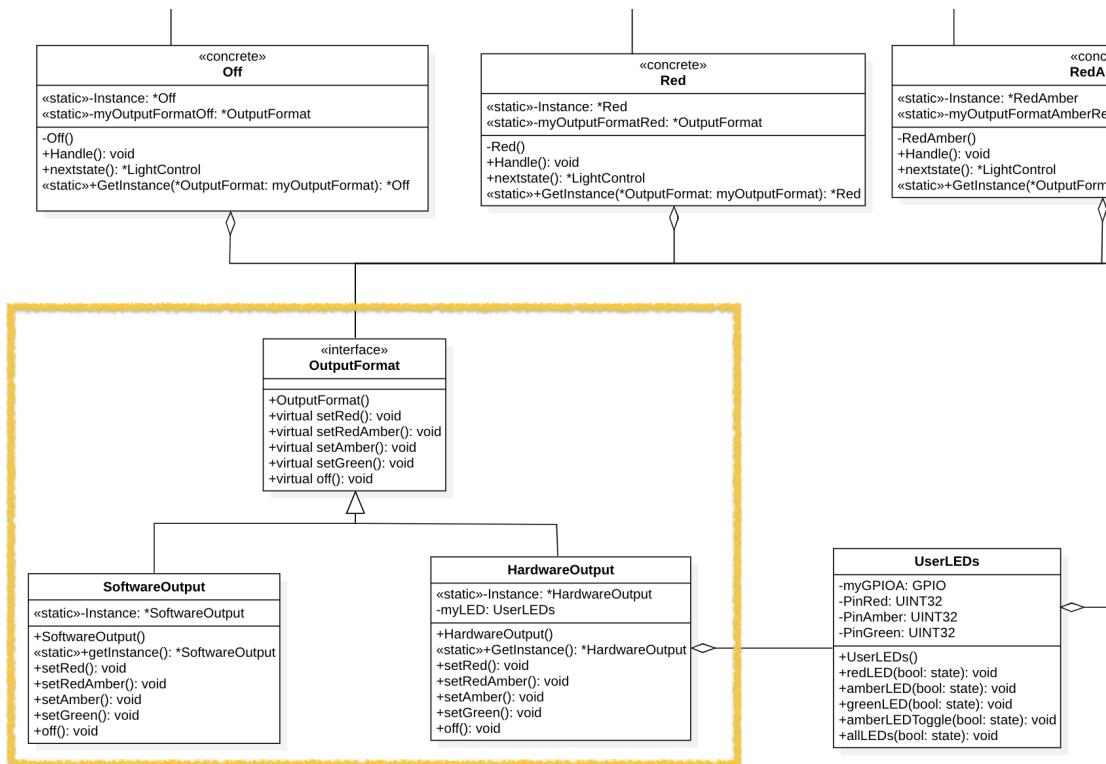


Abbildung 6.1: Auszug aus dem Klassendiagramm - Ausgabeformat

Quelle: Eigene Darstellung

6.2.2 Eingabeformat

Abbildung 6.2 zeigt einen Auszug aus dem Klassendiagramm, welcher für den Wechsel zwischen Hardware- und Softwareeingabe zuständig ist. Auch hier sind eine Interface Klasse „InputFormat“ und zwei erbende Unterklassen „HardwareInput“ und „SoftwareInput“, welche das Interface verwenden, vorzufinden. Der Wechsel zwi-

schen Hardware- und Softwareeingabe funktioniert dabei identisch zur Logik der Hardware- und Softwareausgabe, weshalb darauf an dieser Stelle nicht weiter eingegangen wird.

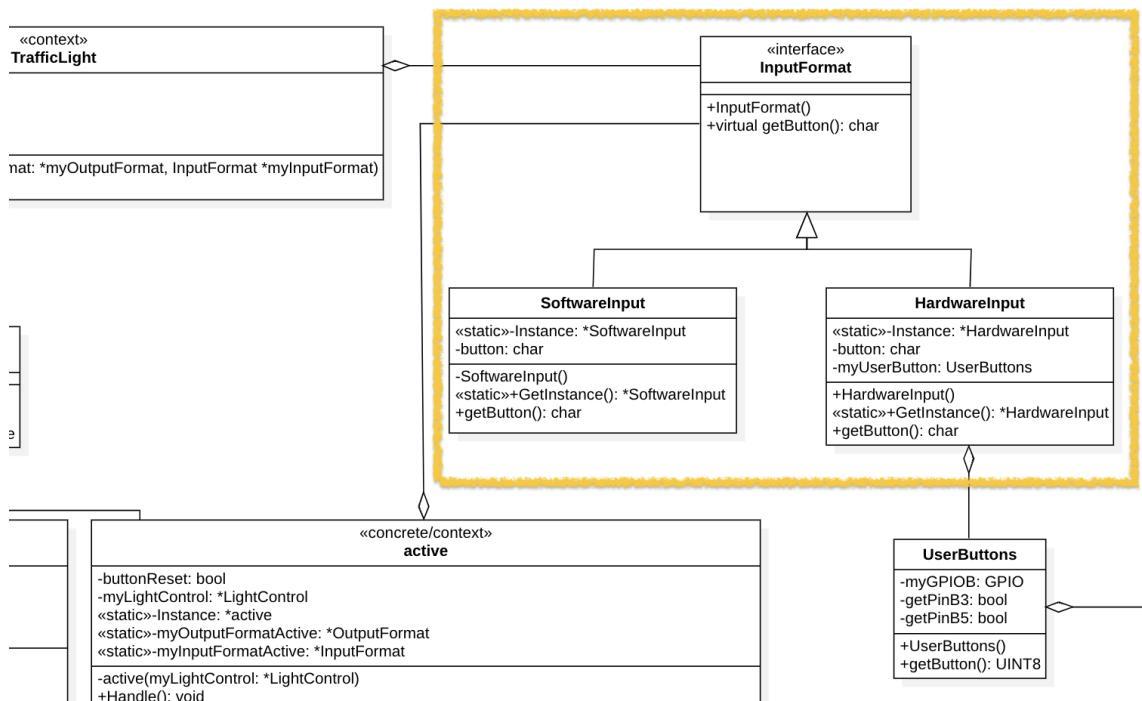


Abbildung 6.2: Auszug aus dem Klassendiagramm - Eingabeformat
 Quelle: Eigene Darstellung

7 Fazit

Schlusslegend kann gesagt werden, dass Ziele dieses Projektes größtenteils erreicht wurden. Die erstellte Ampel kann, sowohl rein software- als auch rein hardwareseitig betrieben werden. Weiterhin ist es möglich den Betriebszustand der Ampel zu wechseln. Ist die Ampel in Betrieb, so wird in der korrekten Farbfolge geschalten, ist sie dagegen außer Betrieb, sollte die gelbe LED blinken. Dieses Ziel konnte leider nicht erreicht werden, da keine vernünftige Methodik gefunden wurde, eine Zeitverzögerung in das Programm einzubauen.

Abbildungsverzeichnis

2.1	Hardwareaufbau der Ampelsteuerung Quelle: Eigene Darstellung	3
3.1	Klassendiagramm der Ampelsteuerung Quelle: Eigene Darstellung	7
3.2	Zustandsautomat der Ampelsteuerung Quelle: Eigene Darstellung	8
4.1	Singleton Patten am Beispiel der Red.h Klasse Quelle: Eigene Darstellung	10
5.1	GPIO Klasse Quelle: Eigene Darstellung	12
5.2	UserLEDs Klasse Quelle: Eigene Darstellung	13
5.3	UserButtons Klasse Quelle: Eigene Darstellung	14
6.1	Auszug aus dem Klassendiagramm - Ausgabeformat Quelle: Eigene Darstellung	18
6.2	Auszug aus dem Klassendiagramm - Eingabeformat Quelle: Eigene Darstellung	19

Tabellenverzeichnis

2.1 Pinbelegung (Hardware/Software)	4
5.1 Rückgabewerte Buttons	15

Quellcodeverzeichnis

4.1	Red.cpp - Singleton in der Klasse Red	11
6.1	main.cpp - Hardwarebetrieb oder Softwarebetrieb	17