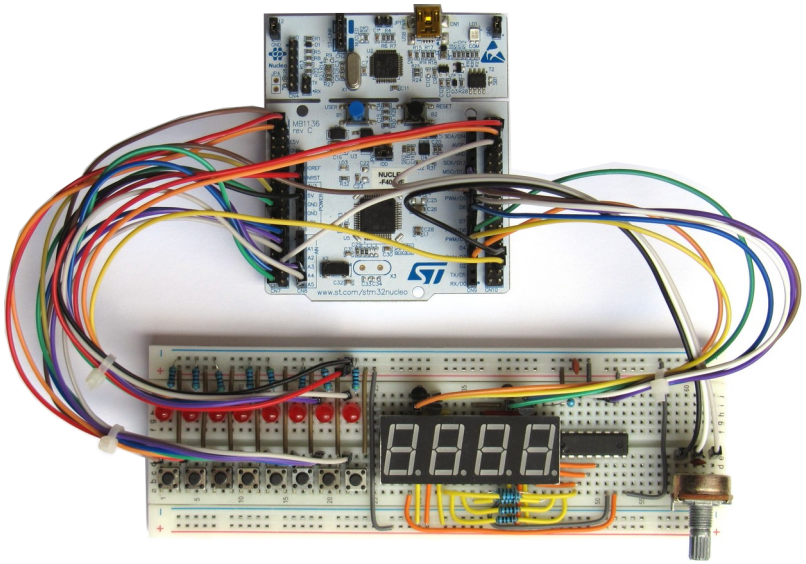
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbeit
FACHBEREICH ELEKTROTECHNIK
UND INFORMATIONSTECHNIK

Mikroprozessoren-Labor



Versuch 1, GPIO

Versuchsdurchführung

Stand: 12. November 2021

I. Aufgaben

1 LED umschalten

In der Vorbereitung haben Sie bereits eine Funktion `task2()` geschrieben, die dafür sorgt, dass eine LED leuchtet, solange der darunter positionierte Taster gedrückt ist.

Schreiben Sie jetzt eine Funktion `task3()`, die bewirkt, dass ein Drücken des Tasters unterhalb einer LED diese LED umschaltet, d. h. ausschaltet, wenn sie eingeschaltet ist, und einschaltet, wenn sie ausgeschaltet ist (englisch: „toggle“).

Diese Funktion kann sehr einfach mit bitweisen Verknüpfungen implementiert werden. Implementieren Sie die Funktion entsprechend dem nachfolgenden UML-Aktivitätsdiagramm.

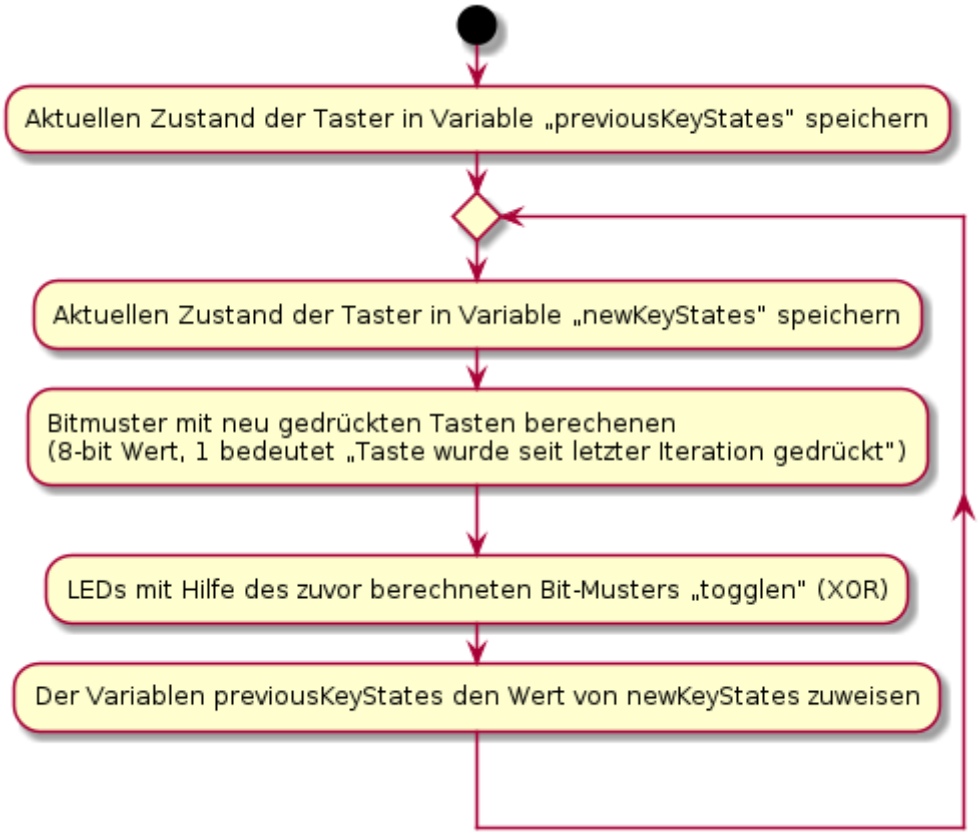


Bild 1: UML-Aktivitätsdiagramm „LEDs umschalten“[1]

Verwenden Sie als Typ der Variablen `previousKeyStates` und `newKeyStates` den passenden Typ aus „stdint.h“.

Die Aktivität „Bitmuster der neu gedrückten Tasten berechnen“ kann mit einer Zeile Code mit einer bitweisen XOR-Verknüpfung, einer bitweisen UND-Verknüpfung und einer bitweisen Negation im-

1 Bitte beachten Sie, dass die Pfeilspitzen in der Mitte der Übergänge nicht UML-konform sind, aber leider von dem hier verwendeten (und abgesehen davon sehr guten) Werkzeug für die Diagrammerstellung hinzugefügt werden.

plementiert werden. Wenn Ihnen der Ausdruck nicht auf Anhieb einfällt, gehen Sie schrittweise vor. Berechnen Sie zunächst in einer Hilfsvariablen `keysPushed` ein Bitmuster, in dem für jede Taste, die seit dem vorangegangenen Schleifendurchlauf gedrückt wurde, eine 1 steht. Diese Information lässt sich aus den Werten von `previousKeyStates` und `newKeyStates` berechnen. Verwenden Sie dann den Wert in `keysPushed` um die LEDs umzuschalten, bei denen der zugeordnete Taster im aktuelle Schleifendurchlauf gedrückt wurde.

Testen Sie die Funktion indem Sie jede Taste mehrmals betätigen. Dabei sollte Ihnen ein „zufälliges Fehlverhalten“ auffallen.

2 Taster entprellen

Es sollte beim Testen das ein oder andere mal passiert sein, dass sich der Zustand der LED scheinbar nicht geändert hat. (Der Effekt ließ sich bei Versuchen mit zwei Aufbauten bei jeweils mindestens zwei Tastern problemlos reproduzieren.) Falls es bei Ihnen immer funktioniert haben sollte, versuchen Sie es evtl. noch einmal, indem Sie die Taster besonders behutsam, also mit gerade so viel Druck, dass sie einschalten, drücken.

Ursache für dieses Verhalten ist das sogenannte „Tastenprellen“. Beim Drücken prallt das bewegliche Element des Tasters unter Umständen mehrfach zurück, bis durch den Druck des Federmechanismus der Kontakt endgültig hergestellt wird. Eine detaillierte Beschreibung finden Sie auf Wikipedia (https://de.wikipedia.org/wiki/Prellen). Von dort wurde auch der nachfolgend dargestellte exemplarische Signalverlauf eines Schließvorgangs übernommen.

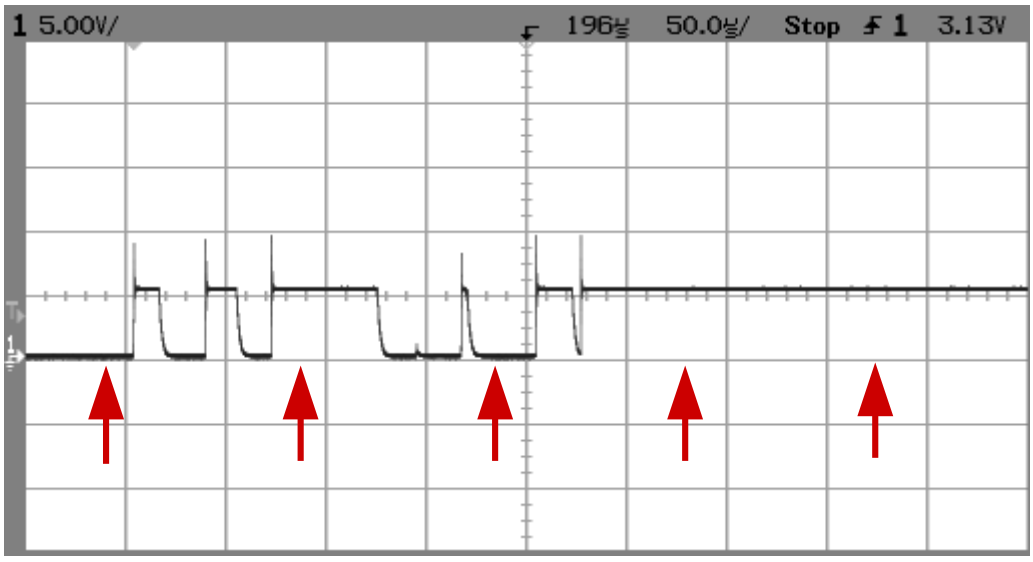


Bild 2: Tastenprellen (Quelle: Wikipedia)

Angenommen, die Abfrage des Eingangssignals erfolgt zu den im Bild eingetragenen Zeitpunkten, würden für einen Tastendruck zwei Wechsel von Low- auf High-Pegel detektiert. Bei unserer Anwendung (LED umschalten) hätte sich der Zustand der LED damit scheinbar nicht geändert.

Eine Software-technische Lösung („Entprellung“) lässt sich relativ leicht implementieren, indem man nach einer Änderung des Eingangswertes eine bestimmte Zeit („Entprellzeit“) wartet, bevor

der Eingang erneut abgefragt wird. Bezogen auf das Bild müsste nach dem mit der zweiten Abtastung entdecken Wechsel mindestens die dritte Abtastung unterdrückt werden. Damit es unter allen Umständen funktioniert, sollten besser zwei Abtastungen unterdrückt werden. Der dargestellte Signalverlauf gehört übrigens zu einem relativ „guten“ Schalter. Bei den einfachen Tastern, wie sie bei dem Laboraufbau verwendet werden, kann die Prellzeit bei mehreren Millisekunden liegen.

Das Startprojekt enthält bereits eine Software-technische Realisierung der Entprellung in Form der Klasse `CDebouncer`. Die Klasse ist eine einfache Variante eines sogenannten „Dekorierers“ (englisch: Decorator). Ein Exemplar einer solchen Klasse wird einem Objekt der „dekorierten“ Klasse (in diesem Fall `BusIn`) quasi „vorangeschaltet“ und stellt unter Nutzung der Methoden der dekorierten Klasse zusätzliche oder veränderte Methoden zur Verfügung.

Im Fall der Dekorierer-Klasse `CDebouncer` liefert die `read`-Methode (und der überladene `operator int ()`) die gleichen Werte wie die gleichnamige Methode der Klasse `BusIn`, allerdings werden die Werte „entprellt“, d. h. nach einer Änderung des Wertes werden für die durch `waitTime` spezifizierte Zeit immer die gleichen Werte geliefert.

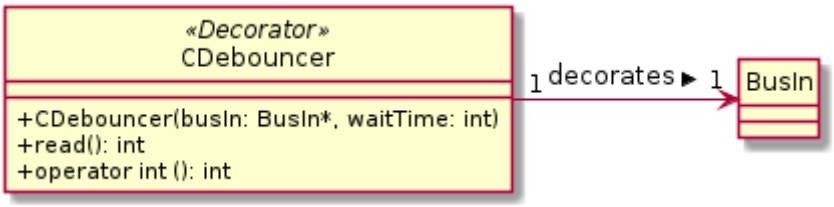


Bild 3: Tastenentpreller `CDebouncer` als Decorator

Schreiben Sie eine Funktion `task4()`, die das gleiche Verhalten wie die Funktion `task3()` implementiert (An-/Ausschalten der LEDs). Greifen Sie für das Lesen der Eingangssignale aber nicht direkt auf das Objekt vom Typ `BusIn` zu. Erzeugen und verwenden Sie vielmehr ein „vorgeschaltetes“ Exemplar der Klasse `CDebouncer` und dessen `read`-Methode (bzw. den überladenen `operator int ()`). Testen sie das Verhalten. Wählen Sie den Wert von `waitTime` so, dass eine zuverlässige Entprellung erreicht wird.

Verwenden Sie in der folgenden Aufgabe (und auch in allen folgenden Laborversuchen) immer die Klasse `CDebouncer` zum Entprellen der Tasten, wenn laut Aufgabenstellung ein Tastendruck eine Aktion auslösen soll.

- **Schriftliche Aufgabe D-1**: Erhöhen Sie die Entprellzeit versuchsweise mal auf 2500 ms und versuchen Sie, die LEDs umzuschalten. Beschreiben Sie das beobachtete Verhalten und begründen Sie, warum zu hohe Entprellzeiten („auf Nummer Sicher gehen“) keine gute Idee sind (vollständige, grammatikalisch korrekte Sätze).
- **Schriftliche Aufgabe D-2**: Schauen Sie sich bei der Nachbereitung der Labors die Implementierung der Klasse `CDebouncer` genau an. Erstellen Sie ein UML-Aktivitätsdiagramm, das die Implementierung der Methode `read()` dokumentiert. Verwenden Sie für das Diagramm Texte in „natürlicher Sprache“, d. h. kopieren Sie nicht den Programmcode sondern beschreiben Sie

die beabsichtigten Aktivitäten bzw. Bedingungen „in Worten“ („==“, „<“, „>=“ etc. dürfen in Bedingungen verwendet werden).

3 Einfaches Lauflicht

3.1 Version 1

Schreiben und testen Sie eine Funktion `task5()` die ein Lauflicht implementiert. „Lauflicht“ bedeutet, dass zunächst nur die LED 0 leuchtet, dann nach 1 Sekunde nur die LED 1, nach einer weiteren Sekunde die LED 2 usw. Nach der LED 7 soll wieder die LED 0 leuchten.

Verwenden Sie für die Implementierung den Schiebeoperator (<<) und die vordefinierte Funktion `thread_sleep_for()`.

3.2 Version 2

Der Nachteil der Version 1 des Programms ist, dass das Programm während der Wartezeit quasi „steht“, d. h. es können keine anderen Aktionen ausgeführt werden. Wollten Sie beispielsweise zusätzlich Eingaben von den Tasten auslesen um die Laufrichtung zu wechseln o. ä., würde ein kurzer Tastendruck während der Wartezeit überhaupt nicht erkannt. Es wäre mit dem Ansatz auch schwierig, LEDs mit unterschiedlichen Frequenzen blinken zu lassen.

Man kann „Warten“ aber auch als „Nebentätigkeit“ realisieren, wenn man die vergangene Zeit messen kann. Angenommen, Sie warten auf das Ende der Vorlesung. Dann schauen Sie auf die Uhr, stellen fest, dass das Ende noch nicht erreicht ist, machen etwas anderes (im Idealfall weiter zuhören), schauen dann mal wieder auf die Uhr usw. bis der Zeitpunkt, auf den Sie warten, erreicht ist.

Etwas näher an der Implementierung könnte „wiederholtes Warten“ entsprechend diesem Ansatz wie im folgenden UML-Aktivitätsdiagramm dargestellt werden.

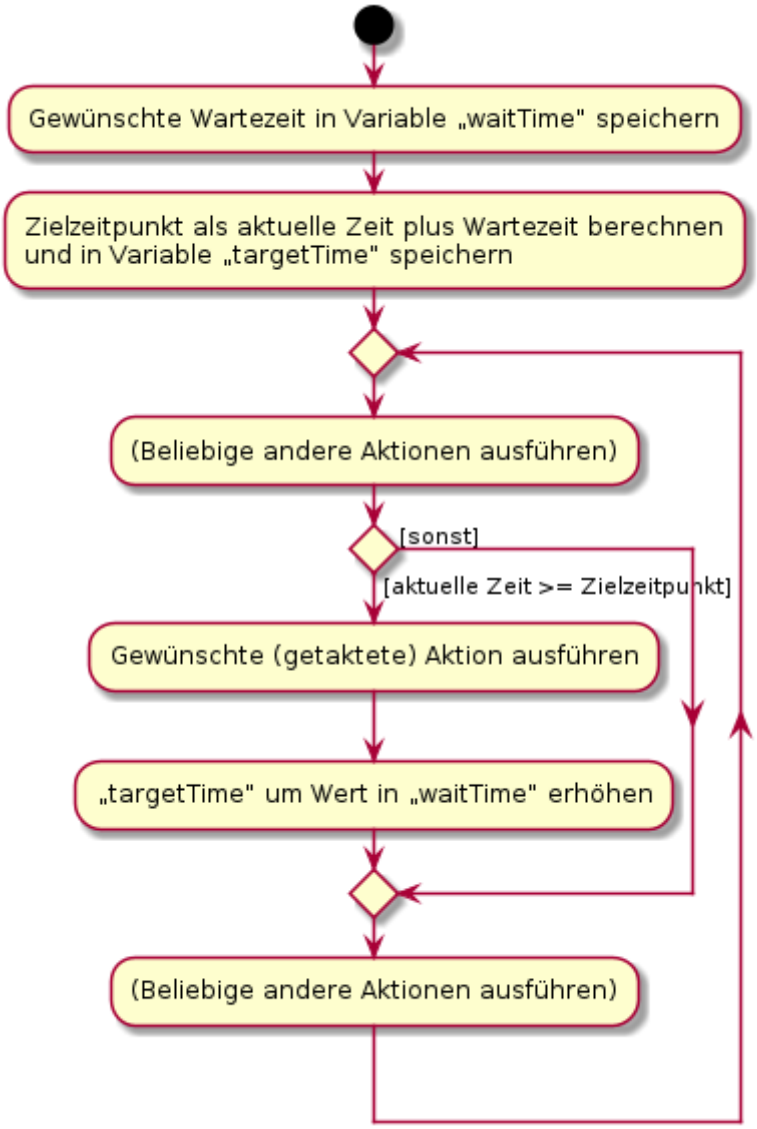


Bild 4: „Aktives Warten“

Die im Diagramm eingezeichneten Aktivitäten „(Beliebige andere Aktion ausführen)“ sind „Platzhalter“ (deshalb in Klammern). Sie können an diesen Stellen in Ihrem Programm zusätzliche Aktionen ausführen, bei der aktuellen Aufgabenstellung gibt es an diese Stellen allerdings (noch) nichts zu tun.

Schreiben und testen Sie eine Funktion `task6()` die das gleiche Verhalten wie die Funktion `task5()` implementiert (Lauflicht), das Warten aber nicht durch Aufruf der Funktion `thread_sleep_for()` sondern entsprechend dem obigen UML-Aktivitätsdiagramm realisiert.

Um den aktuellen Zeitpunkt (in Millisekunden seit Programmstart) zu ermitteln, kopieren Sie den Code aus der Implementierung von `CDebouncer::millisSinceStart`. Der – zugegebener Maßen etwas kompliziert aussehende – Ausdruck
„`std::chrono::duration_cast<std::chrono::milliseconds>(HighResClock::now(`

`).time_since_epoch()).count()`“ liefert Ihnen die Millisekunden seit dem Start des Programms als `int64_t`, und diesen Wert können Sie als „aktuelle Zeit“ verwenden.

- **Schriftliche Aufgabe D-3**: Immer, wenn man mit Zeiten arbeitet, sollte man sich darüber klar sein, bis zu welchem Zeitpunkt der Wertebereich der verwendeten Variablen ausreicht (vergl. z. B. https://medium.com/@brigitaplika/integer-overflow-or-when-255-1-0-can-cause-problems-b76f6c848920). Geben Sie in Ihrer Labordokumentation (einschließlich Rechenweg) an, wie viele Jahre das Programm gelaufen sein wird, wenn der von dem obigen Ausdruck gelieferte Wert den Maximalwert eines `int64_t` erreicht.
- **Schriftliche Aufgabe D-4**: Angenommen, Sie würden, um Speicherplatz zu sparen, die Variable `targetTime` als `uint32_t` deklarieren. Wie viele Jahre könnte Ihr Programm dann laufen bis ein Überlauf auftritt? (Rechenweg angeben)

3.3 Version 3

Insbesondere wenn man mehrere Wartezeiten verwalten muss, wird das Hauptprogramm mit diesem Ansatz schnell unübersichtlich. Man benötigt für jede verwaltete Wartezeit zwei Variablen und muss sich – neben der eigentlich gewünschten Aktion – beim Erreichen des Zielzeitpunkts um die Berechnung des neuen Wertes für den Zielzeitpunkt kümmern.

Deshalb ist es sinnvoll, die Timer-Funktionalität in eine Klasse auszulagern. Die beiden benötigten Variablen (`targetTime` und `waitTime`) werden zu Attributen der Klasse. Die Initialisierung erfolgt im Konstruktor. Und der Vergleich mit der aktuellen Zeit (und bei Bedarf die Aktualisierung von `targetTime`) erfolgt in der Methode `timeReached()`, die auch die Information liefert, ob der Zielzeitpunkt beim Aufruf erreicht war.

Die Klasse ist im Startprojekt bereit angelegt. Ergänzen Sie die Implementierung der Methoden. Testen Sie Ihre Implementierung anschließend, indem Sie in `main` die bereits vorhandene Funktion `task7()` auskommentieren und aufrufen.

Schreiben und testen Sie eine Funktion `task8()` die das gleiche Verhalten wie die Funktion `task6()` implementiert (Lauflicht), aber statt der Implementierung des Timer-Verhaltens als Teil der `main`-Funktion ein Objekt `chasingLightTimer` vom Typ `CPolledTimer` verwendet.

4 Konfigurierbares Lauflicht

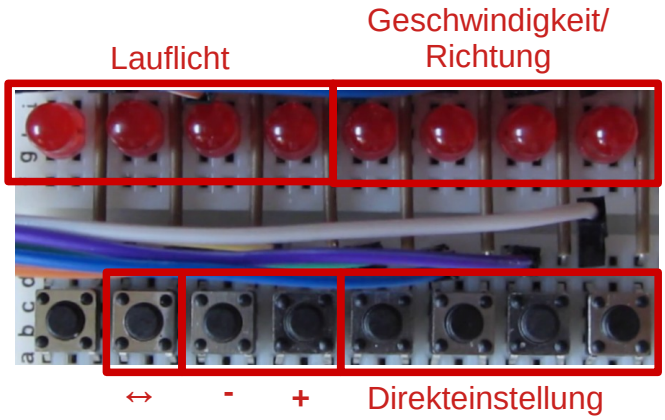
Mit den Vorarbeiten aus den vorangegangenen Aufgaben haben Sie die notwendigen Komponenten zur Verfügung, um in der Endlosschleife einer `taskN()`-Funktion sowohl die Taster abzufragen[2] als auch periodisch Aktionen (z. B. das Weiterschalten des Lauflichts) auszuführen.

Implementieren Sie jetzt mit Hilfe dieser Komponenten eine Funktion `task9()`, die die Anwendung „konfigurierbares Lauflicht“ realisiert. Bei dieser Anwendung soll das Verhalten eines Lauflichts über den Wert einer Variablen gesteuert werden. Die Variable kann die Werte -7 bis +7 annehmen. Ein negativer Wert bedeutet, dass das Lauflicht von rechts nach links läuft, ein positiver Wert,

2 Selbstverständlich unter Nutzung der Entpreller-Klasse!

dass es von links nach rechts läuft. Der absolute Wert der Variablen bestimmt die Zeit zwischen zwei Zustandswechseln nach der Formel „absoluter Wert * 250 ms“ und damit die Geschwindigkeit des Lauflichts. Ist der Wert der Steuervariablen 0, ist das Lauflicht ausgeschaltet (keine LED leuchtet).

Die LEDs und Taster werden entsprechend dem nachfolgenden Bild in Gruppen aufgeteilt.



Die LED-Gruppe „Lauflicht“ stellt das Lauflicht dar. Die Gruppe „Geschwindigkeit/Richung“ stellt den Wert der Steuervariablen als vorzeichenbehaftete 4-bit Binärzahl (Zweierkomplement) dar. Mit den Tastern der Gruppe „Direkteinstellung“ können die darüber liegenden LEDs getoggelt werden. Damit kann der Wert der Steuervariablen direkt eingestellt werden. Verwenden Sie für die Auswertung dieser vier Taster den gleichen Ansatz wie in `task3`.

Die Taste „+“ erhöht den Wert der Steuervariablen um 1, die Taste „-“ vermindert ihn um 1. Hat die Variable bereits den Minimal- oder Maximalwert, hat ein weiterer Tastendruck keine Auswirkung. Die Taste „↔“ wechselt die Laufrichtung, ohne die Geschwindigkeit zu verändern.

Gehen sie bei der Bearbeitung der Aufgabe schrittweise vor. Implementieren Sie zunächst die Anzeige des Werts der Steuervariable mit den vier rechten LEDs.[3] Implementieren Sie dann die Direkteinstellung und anschließend die Funktion der Tasten „+“, „-“ und „↔“. Schließlich ergänzen Sie das eigentliche Lauflicht. Für die Änderung der Wartezeit ergänzen Sie in der Klasse `CPolledTimer` eine Methode „`void updateWaitTime(int32_t waitTime)`“. Sie aktualisiert den Wert des Attributs `m_waitTime` und berechnet einen neuen Wert für `m_targetTime`. Vergessen Sie nicht, die Deklaration der Methode in der Header-Datei entsprechend den Vorgaben zu dokumentieren.

Zusatzfunktion 1 (freiwillig): Falls Sie noch Zeit haben, können Sie den noch nicht verwendeten Taster ganz links einsetzen, um zwischen einem „Einstellungsmodus“ (Verhalten wie oben beschrieben) und einem „Produktionsmodus“ zu wechseln. Im „Produktionsmodus“ haben alle Taster (außer dem ganz links) keine Funktion und das Lauflicht läuft (mit dem im Einstellungsmodus konfigurierten Verhalten) über alle 8 LEDs.

3 Die Gruppierung der LEDs und Taster könnte hier sehr gut durch mehrere BusIn- und BusOut-Objekte abgebildet werden. Für diese Aufgabe verwenden Sie aber bitte die bereits in den vorangegangenen Aufgaben benutzten Objekte mit jeweils 8 GPIO-Pins, da Sie in dieser Aufgabe u. a. die Verwendung der Bit-Operatoren üben sollen.

Zusatzfunktion 2 (freiwillig): Der Wechsel in einen Einstellungsmodus wird oft abgesichert, damit er nicht versehentlich erfolgt. Ändern Sie das Verhalten so, dass der Wechsel vom Produktionsmodus zum Einstellungsmodus nur erfolgt, wenn der linke Taster mindestens 3 Sekunden gedrückt wird.