Skriptum Mikrocontroller

Teil 4 Programmiertechnik

DI(FH) Andreas Pötscher, HTL Litec

1.9.2025

Inhaltsverzeichnis

Verknüpfungssteuerungen **2**
Verknüpfungssteuerungen Programmaufbau 2
Präprozessor 3
Wahrheitstabellen 5

Ablaufsteuerungen **6**
State Machine 6
Ablaufsteuerungen Programmaufbau 7
Ablauf des Programmes 8
Signalflanken 9

Copyright- und Lizenz-Vermerk: Das vorliegende Werk kann unter den Bedingungen der Creative Commons License CC-BY-SA 3.0, siehe http://creativecommons.org/licenses/by-sa/3.0/deed.de, frei vervielfältigt, verbreitet und verändert werden. Eine kurze, allgemein verständliche Erklärung dieser Lizenz kann unter http://creativecommons.org/licenses/by-sa/3.0/deed.de gefunden werden. Falls Sie Änderungen durchführen, dokumentieren Sie diese im folgenden Änderungsverzeichnis:

Datum	Beschreibung der durchgeführten Änderung	Autor
2022	V1.0 . . . Neuerstellung des Dokuments	Andreas Pötscher, HTL Linz–Paul-Hahn-Straße (LiTec)
29.08.2024	V1.1 . . . Übernahme in Markdown und allgemeine Überarbeitung	Andreas Pötscher, HTL Linz–Paul-Hahn-Straße (LiTec)

Verknüpfungssteuerungen

Verknüpfungssteuerungen sind die einfachsten Steuerungen. Dabei wird nur aufgrund von Eingangszuständen ein Ausgangszustand eingestellt. Die Reihenfolge oder die zeitliche Abfolge von Eingängen hat keinen Einfluss auf die Ausgänge.

Abbildung 1: Verknüpfungssteuerungen

Verknüpfungssteuerungen Programmaufbau

Das Programm kann dabei immer nach folgendem Schema einfach aufgebaut werden.

1. Initialisierungsteil

Im Initialisierungsteil werden alle die GPIOs je nach Aufgabenstellung als Ein- und Ausgänge konfiguriert. Wird ein interner PullUp Widerstand benötigt, wird dieser hier aktiviert. Der Initialisierungsteil wird einmal beim Start vor der `while(1)` Schleife ausgeführt.

```
int main()
{
    //Initialisierungsteil
    //Hier werden die GPIOs als Ein- und Ausgänge initialisiert
    DDRA = 0x06;
    DDRB = 0x00;
```

2. Eingänge einlesen

In der `while(1)` Schleife wird dann die eigentliche Programmlogik implementiert. Dabei müssen als erstes immer alle Eingänge eingelesen werden. Dies ist notwendig damit sich während der Logikauswertung danach die Werte nicht ändern können. Wenn sich ein Eingangswert ändert (z.B. ein Taster gedrückt wird), wird dies erst im nächsten Schleifendurchlauf berücksichtigt.

```
while(1)
{
    //1. Eingänge Lesen
    int tasterA = PINB >> 3 & 0x01;
```

3. Ausgänge setzen

Hier werden, nach der geforderten Logik, aus den Eingängen die Werte der Ausgänge ermittelt. Meistens werden hier `if else` oder `switch case` Verzweigungen verwendet. Wichtig ist dabei, dass in jeder Verzweigung immer alle Ausgänge gesetzt werden. Wird zum Beispiel eine LED im `if` eingeschaltet und im `else` nicht ausgeschaltet schaltet sich die LED nicht aus, da das dementsprechende Bit im PORT-Register immer noch auf 1 gesetzt bleibt.

```
//2. Ausgänge setzen
if(tasterA)
{
    PORTA |= 0x01 << 2;
    PORTA &= ~(0x01 << 1);
}
else
{
    //Es sollten in jeder Verzweigung alle Ausgänge gesetzt werden.
    PORTA |= 0x01 << 1;
    PORTA &= ~(0x01 << 2);
}
```

Alles zusammen sieht dann so aus:

```
int main()
{
    //Initialisierungsteil
    //Hier werden die GPIOs als Ein- und Ausgänge initialisiert
    DDRA = 0x06;
    DDRB = 0x00;

    while(1)
    {
        //1. Eingänge Lesen
        int tasterA = PINB >> 3 & 0x01;
        //2. Ausgänge setzen
        if(tasterA)
        {
            PORTA |= 0x01 << 2;
            PORTA &= ~(0x01 << 1);
        }
        else
        {
            //Es sollten in jeder Verzweigung alle Ausgänge gesetzt werden.
            PORTA |= 0x01 << 1;
            PORTA &= ~(0x01 << 2);
        }
    }
}
```

Präprozessor

Zur Vereinfachung und Vermeidung von Redundanzen können die verwendeten Pins und das Ansteuern der Ausgänge mit Präprozessormakros programmiert werden. Diese beginnen immer mit `#define`. Der Präprozessor kopiert den Code vor dem eigentlichen Kompilieren an die richtigen Stellen.

Z.B. Code mit Präprozessormakro

```
#define PINLED1 1
PORTA |= 0x01 << PINLED1;
```

Beim Übersetzten des Programmes wird dabei als erstes der Präprozessor ausgeführt. Der Code sieht dann so aus.

```
PORTA |= 0x01 << 1;
```

Der Präprozessor führt dabei einfaches Suchen und Ersetzen aus. Damit werden zwei Vorteile erzielt:

- Die Lesbarkeit des Codes wird deutlich erhöht
- Soll ein Pin verändert werden (z.B. wird eine LED an einem anderen PIN angeschlossen) muss der Code nur an einer Stelle verändert werden.

Das vorherige Programm sieht mit Präprozessor-Makros so aus.

```
#define PINLED1 1
#define PINLED2 2
#define PINTASTERA 3

#define LED1ON PORTA |= 0x01 << PINLED1
#define LED1OFF PORTA &= ~(0x01 << PINLED1)
#define LED2ON PORTA |= 0x01 << PINLED2
#define LED2OFF PORTA &= ~(0x01 << PINLED2)

int main()
{
    //Initialisierungsteil
    //Hier werden die GPIOs als Ein- und Ausgänge initialisiert
    DDRA = 0x06;
    DDRB = 0x00;

    while(1)
    {
        //1. Eingänge Lesen
        int tasterA = PINB >> PINTASTERA & 0x01;
        //2. Ausgänge setzen
        if(tasterA)
        {
            LED1ON;
            LED2OFF;
        }
        else
        {
            //Es sollten in jeder Verzweigung alle Ausgänge gesetzt werden.
           LED2ON;
           LED1OFF;
        }
    }
}
```

Wahrheitstabellen

Die Logik für eine Verknüpfungssteuerung lässt sich sehr gut mit einer Wahrheitstabelle darstellen. Dabei wird für jeden Ein- und Ausgang eine Spalte und für jede Eingangskombination eine Zeile angelegt. Bei drei Sensoren (S1 bis S3) und zwei Leuchten (L1 und L2) sieht das dann folgendermaßen aus:

S1	S2	S3	L1	L2
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Wenn z.B. Leuchte2 leuchten soll wenn alle 3 Sensoren logisch 1 melden und Leuchte1 leuchten soll wenn 1 oder 2 von den 3 Sensoren logisch 1 meldet sieht das so aus:

S1	S2	S3	L1	L2
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

Oder als Source Code (nur Logikteil)

```
//1. Eingänge Lesen
uint8_t S1 = PINB >> PINTASTER1 & 0x01;
uint8_t S2 = PINB >> PINTASTER2 & 0x01;
uint8_t S3 = PINB >> PINTASTER3 & 0x01;

//2. Ausgänge setzen
if(S1 && S2 && S3)
{
   LED1OFF;
   LED2ON;
}
else if(S1 || S2  || S3)
{
    LED1ON;
    LED2OFF;
}
else
```

```
{
    LED1OFF;
    LED2OFF;
}
```

Ablaufsteuerungen

Bei Auflaufsteuerungen werden nicht nur Eingangszustände sondern auch interne Zustände berücksichtig. Damit sind Abfolgesteuerungen möglich.

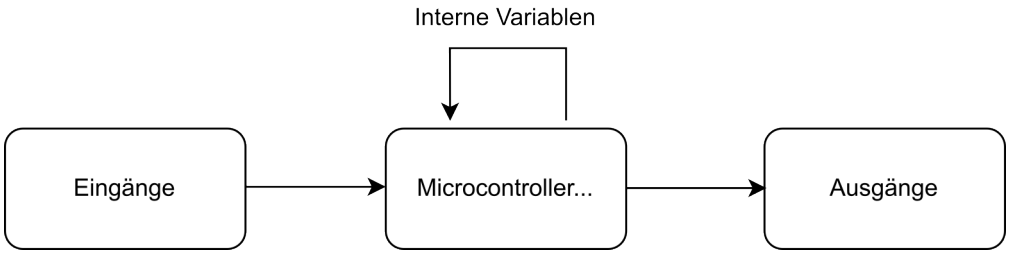

Abbildung 2: Ablaufsteuerungen

Damit kann zum Beispiel eine Steuerung mit einem Ein- und einem Austaster realisiert werden. Ein Led lässt sich dann nur einschalten, wenn sich die Steuerung im Ein-Modus befindet.

State Machine

In der Informatik kann dieses Verhalten mit einer State Machine (Zustandsautomat) beschrieben werden. Dabei werden Zustände und Übergänge zwischen diesen beschrieben. Ein einfaches Beispiel mit 2 Zuständen und 2 übergängen sieht so aus.

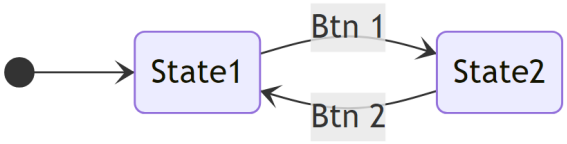

Abbildung 3: Statemachine

Dabei gibt es zwei Zustände (*State1* und *State2*). Durch den Kreis links wird der Startzustand dargestellt. Das heißt bei Programmstart befindet sich das Programm in *State1*. Mit den Eingänge *Btn1* und *Btn2* kann zwischen diesen Zuständen gewechselt werden.

In einem Beispiel in dem eine Steuerung ein- und ausgeschaltet werden soll, werden zwei Zustände benötigt: *controllOn* und *controllOff*. Dabei kann mit 2 Tastern (*btnOn* und *btnOff*) zwischen den States gewechselt werden. Beim Programmstart soll im *controllOff state* gestartet werden. Als State Diagramm sieht das ganz so aus.

In der Software wird der State in einer eigenen Variablen gespeichert. Jeder Wert der Variable steht dann für einen State. In diesem Beispiel:

- *controllOff* 0
- *controllOn* 1

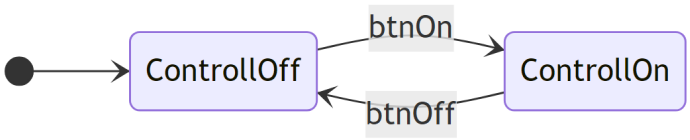

Abbildung 4: Statemachine Beispiel

Ablaufsteuerungen Programmaufbau

Das Programm für das Beispiel kann dabei nach folgendem Schema einfach aufgebaut werden.

1. Initialisierungsteil

Im Initialisierungsteil werden alle die GPIOs je nach Aufgabenstellung wie im vorherigen Kapitel beschrieben initialisiert.

```
int main()
{
    DDRA = 0x02;
    DDRB = 0x00;
```

2. Initialsieren der Variable in der der Zustand (State) gespeichert wird.

Im zweiten Schritt muss dann eine Variable für den Zustand initialisiert und deklariert werden. Die Variable kann 2 Werte (Zustände) annehmen:

- **0** Steuerung ist ausgeschaltet
- **1** Steuerung ist eingeschaltet

Die Variable wird beim Start auf den Wert 0 gesetzt. Damit ist die Steuerung in ausgeschaltetem Zustand. Um Redundanzen zu vermeiden sollte immer nur eine Variable für den Zustand verwendet werden.

```
    uint8_t controllOn = false;
```

3. Eingänge einlesen

In der `while(1)` Schleife werden wie, im vorherigen Kapitel beschrieben, immer alle Eingänge gelesen.

```
while(1)
{
    uint8_t btnOn = PINB >> PINBTNON & 0x01;
    uint8_t btnOff = PINB >> PINBTNOFF & 0x01;
    uint8_t btnLed = PINB >> PINBTNLED & 0x01;
```

4. Setzen der Zustandsvariablen. Nachdem die Eingänge gelesen wurden muss im Programm überprüft werden ob sich der Zustand ändern muss. In diesem Beispiel werden die Taster btnOn und btnOff abgefragt, um die Variable für den Zustand (controlOn) entsprechend zu setzen. Zusätzlich wird hier noch eine Led, die den Zustand anzeigt, gesetzt oder gelöscht.

```
if(btnOn)
{
    controllOn = true;
    LED2ON;
}
if(btnOff)
{
    controllOn = false;
    LED2OFF;
}
```

5. Ausgänge setzen

Hier werden, wie im vorherigen Kapitel beschrieben, alle Ausgänge gesetzt. Im Unterschied müssen jetzt aber nicht nur die Eingänge sondern auch der Zustand mit berücksichtigt werden. Im Beispiel lässt sich die Led mit dem Taster *ledOn* nur Einschalten wenn das Programm im Zustand controllOn ist.

```
if(btnLed && controllOn)
{
    LED1ON;
}
else
{
    LED1OFF;
}
```

Ablauf des Programmes

Wie man an dem Beispiel sieht kommt dabei das Ergebnis nicht nur auf die Zustände der Eingänge an, sondern auch auf die Reihenfolge in der diese Betätigt werden. Werden z. B. folgende Eingänge in der gegeben Reihefolge gesetzt:

```
1. PINB2 (btnLed) = true
```

Ist die LED an PA1 **nicht** eingeschaltet. Der Zustand bleibt auf controllOff.

Werden aber die Eingänge in folgender Reihenfolge betätigt.

```
1. PINB0 (btnOn) = true
2. PINB0 (btnOn) = false
3. PINB2 (btnLed) = true
```

ist die LED an PA1 eingeschaltet. Dadurch, dass der Eingang PINB0 auf true gesetzt wird, wird interne Variable `controlOn` auf true gesetzt. Das Ergebnis wenn PINB2 auf true gesetzt wird verändert sich dadurch.

Diese zeitliche Ablauf kann in einem Signal-Zeit Diagramm dargestellt werden.

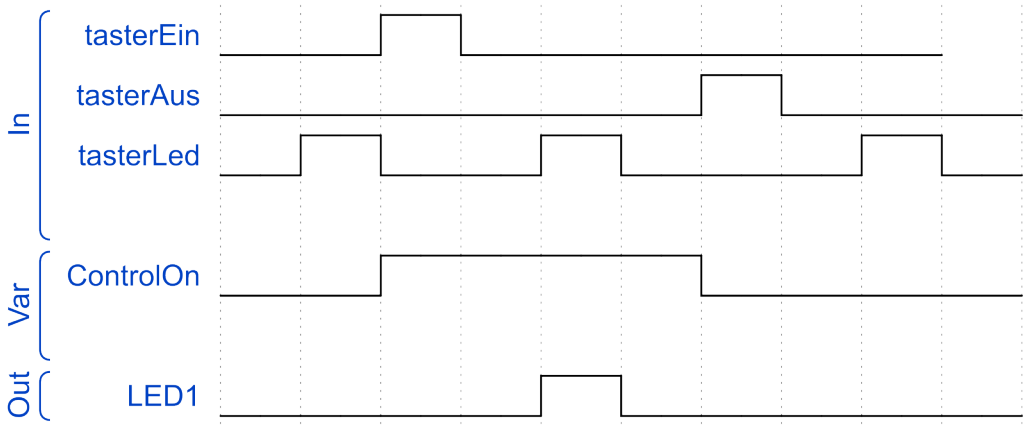

Abbildung 5: Signal Zeitverlauf

In diesem Fall wird die zeitliche Abfolge aller Eingänge (In), Variablen (Var) und Ausgänge (Out) dargestellt.

Signalflanken

Wird mit einem Microcontroller ein Eingang z.B. ein Taster abgefragt, kann durch die zyklische Abfrage nicht genau bestimmt werden wie oft und wann der Taster gedrückt wurde.

In folgendem Beispiel kommt es darauf an wie lange der Taster betätigt wurde ob die LED am Ende eingeschaltet ist oder nicht.

```
while(1)
{
    uint8_t btn = PINB >> PINBTN & 0x01;
    if(btn)
    {
        if(ledOn == 0)
        {
        ledOn = 1;
        LEDON;
        }
        else
        {
        ledOn = 0;
        LEDOFF;
        }
    }
}
```

Praktisch wird sich die LED sehr schnell Ein- und Ausschalten. Dieses Verhalten kann wieder sehr gut mit einer Statemachine beschrieben werden.

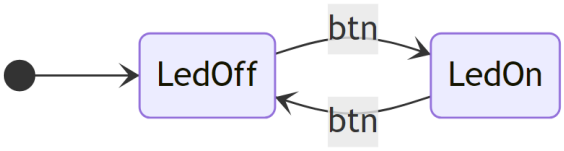

Abbildung 6: Statemachine Flanken

Da das gleiche Signal für den Zustandswechsel verwendet wird, wechselt bei jedem Schleifendurchlauf der Zustand. Für solche Steuerungsaufgaben ist es notwendig die Änderung des Eingangs zu detektieren. Z.B. Wenn im vorigen Beispiel die Led genau einmal umschaltet werden soll.

Dazu ist der zeitliche Signalverlauf des Tasters wichtig.

Abbildung 7: Signal Zeitverlauf Taster

Dabei wird der bei einmaligem Drücken des Tasters fünfmal ein High Pegel eingelesen. Die Led würde also um 5 mal umschalten. Damit das nur einmal passiert werden die Flanken des Signals verwendet. Es gibt eine steigende und eine fallende Flanke.

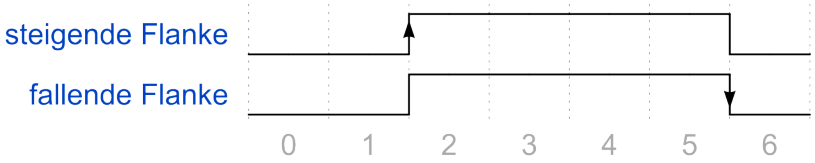

Abbildung 8: Steigende und fallende Flanke

Die steigende Flanke tritt auf wenn sich das Signal von Low auf High ändert. Also wenn der Taster gedrückt wird. Die fallende Flanke tritt auf wenn sich das Signal von High auf Low ändert. Also wenn der Taster losgelassen wird. Damit die Flanken im Programm detektiert werden können muss der Wert des Eingangs beim letzten Durchlauf gespeichert werden.

Dazu wird vor der `while(1)` Schleife eine Variable deklariert. Am Ende der Schleife wird der Wert des Eingangs in die Variable gespeichert.

```
uint8_t lastBtn = 0;
while(1)
{
    //Hier können die Flanken detektiert werden
    lastBtn = btn;
}
```

Als Diagramm sehen die Variablen btn und lastBtn wie folgt aus:

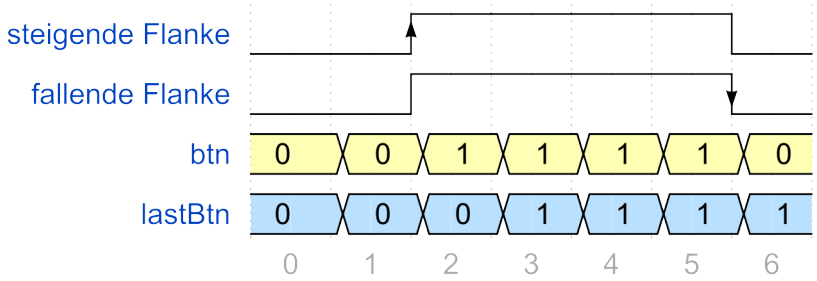

Abbildung 9: Werte der Variablen btn und lastBtn

Bei einer steigenden Flanke (*engl. rising Edge*) ist der aktuelle Wert `btn` dann 1 und der vorherige Wert `lastBtn` dann 0. Als Code sieht das dann so aus.

```
uint8_t lastBtn = 0;
while(1)
{
    uint8_t btn = PINB >> PINBTN & 0x01;
    if(btn == 1 && lastBtn == 0)
    {
        //Steigende Flanke
    }
    lastBtn = btn;
}
```

Bei einer fallenden Flanke (*engl. falling Edge*) ist der aktuelle Wert `btn` dann 0 und der vorherige Wert `lastBtn` dann 1. Als Code sieht das dann so aus.

```
uint8_t lastBtn = 0;
while(1)
{
    uint8_t btn = PINB >> PINBTN & 0x01;
    if(btn == 0 && lastBtn == 1)
    {
        //Fallende Flanke
    }
    lastBtn = btn;
}
```

Als dritte Möglichkeit kann auch jede beliebige Flanke (*engl. any Edge*) einfach detektiert werden. Dazu müssen die Werte von `btn` und `lastBtn` ungleich sein.

```
uint8_t lastBtn = 0;
while(1)
{
    uint8_t btn = PINB >> PINBTN & 0x01;
    if(btn != lastBtn)
    {
        //Beliebige Flanke
    }
    lastBtn = btn;
}
```

Das Programm mit dem eine Led mit einem Taster umgeschaltet werden kann sieht dann wie folgt aus.

```
uint8_t ledOn = 0;
uint8_t lastBtn = 0;

while(1)
{
    uint8_t btn = PINB >> PINBTN & 0x01;

    if(btn == 1 && lastBtn == 0)
    {
        if(ledOn == 0)
        {
        ledOn = 1;
        LEDON;
        }
        else
        {
        ledOn = 0;
        LEDOFF;
        }
    }

    lastBtn = btn;
}
```