Skriptum Mikrocontroller

Teil 4 Programmiertechnik

DI(FH) Andreas Pötscher, HTL Litec

Inhaltsverzeichnis

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 im Markdown und allgemeine Überarbeitung	Andreas Pötscher, HTL Linz–Paul-Hahn-Straße (LiTec)

Verknüpfungsteuerungen

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

Verknüpfungsteuerungen 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 Wiederstand benötigt wird dieser hier aktiviert. Der Initialisierungsteil wird einmal beim Start vor der `while(1)` Schleife ausgeführt.

```
int main()
{
    //Initialsierungsteil
    //Hier werden die GPIOs als Ein- und Ausgänge initialsiert
    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 Eingänge 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()
{
    //Initialsierungsteil
    //Hier werden die GPIOs als Ein- und Ausgänge initialsiert
    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()
{
    //Initialsierungsteil
    //Hier werden die GPIOs als Ein- und Ausgänge initialsiert
    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 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
int S1 = PINB >> PINTASTER1 & 0x01;
int S2 = PINB >> PINTASTER2 & 0x01;
int 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.

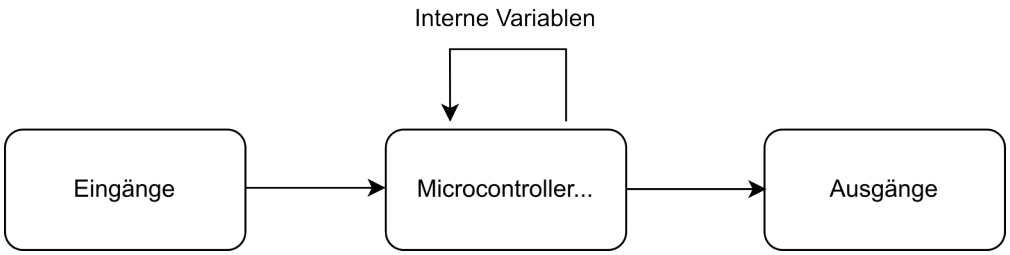

Abbildung 2: Ablaufsteuerungen

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.

```
#define LED1ON PORTA |= 0x01 << 1
#define LED1OFF PORTA &= ~(0x01 << 1)

int main()
{
    //Initialsierungsteil
    //Hier werden die GPIOs als Ein- und Ausgänge initialsiert
    DDRA = 0x02;
    DDRB = 0x00;

    //Variablen für die internen Zustände
    bool controllOn = false;

    while(1)
    {
        //1. Eingänge Lesen
        int tasterEin = PINB >> 0 & 0x01;
        int tasterAus = PINB >> 1 & 0x01;
        int tasterLed = PINB >> 2 & 0x01;

        //2. Interne Variablen setzen
        if(tasterEin)
        {
             controllOn = true;
        }
        if(tasterAus)
```

```
        {
            controllOn = false;
        }

        //3. Ausgänge setzen
        if(tasterLed && controllOn)
        {
            LED1ON;
        }
        else
        {
            LED1OFF;
        }
    }
}
```

Dabei kommt 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 = true
```

Ist die LED an PA1 **nicht** eingeschaltet.

Werden aber die Eingänge in folgender Reihenfolge betätigt.

```
1. PINB0 = true
2. PINB0 = false
3. PINB2 = 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.

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

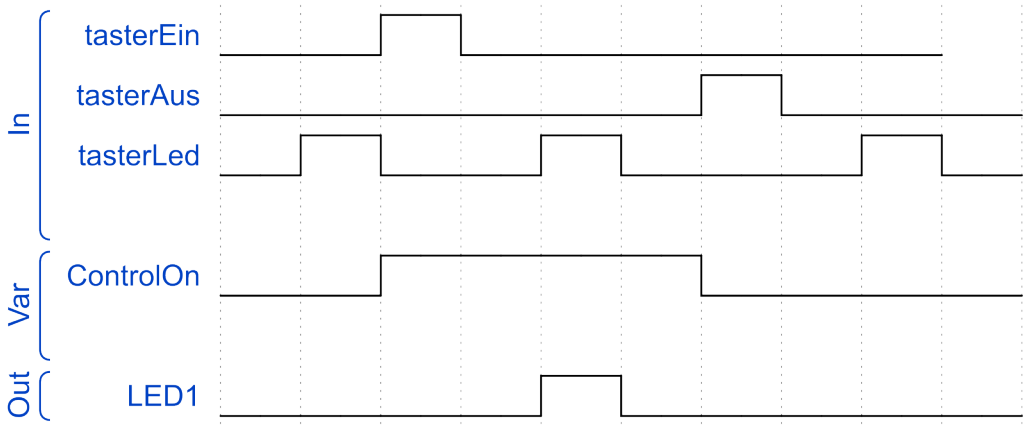

Abbildung 3: Signal Zeitverlauf

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

Flankensteuerungen

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 auf welchen Wert die Counter Variable hochgezählt wird.

```
int counter = 0;
while(1)
{
    int btn = PIND >> PIN_BTN & 0x01;
    if(btn)
    {
        counter++;
    }
}
```

Für manche Steuerungsaufgaben ist es notwendig bei Änderung eines Eingangs genau diese Änderung zu detektieren. Z.B. Wenn wir im vorigen Beispiel zählen wollen wie oft der Taster gedrückt wurde.

Sehen wir uns dazu den zeitlichen Signalverlauf des Tasters an.

Abbildung 4: Signal Zeitverlauf Taster

Dabei wird der bei einmaligem Drücken des Tasters 5 mal ein High Pegel eingelesen. Die Zähler würde also um 5 hochzählen. Damit nur einmal gezählt wird können wir die Flanken des Signals verwenden. Es gibt eine steigende und eine fallende Flanke.

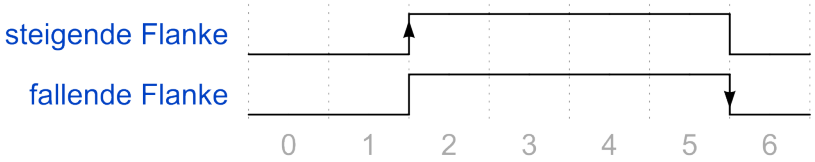

Abbildung 5: Steigende und fallende Flanke

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 verwendet werden können muss der Wert des Eingangs beim letzen Durchlauf gespeichert und mit dem aktuellen Wert verglichen werden.

```
int counter = 0;
//Varible vor dem while(1) loop deklarieren
int lastBtn = 0;
while(1)
{
    int btn = PIND >> PIN_BTN & 0x01;

    //Abfragen der steigenden Flanke
    if(btn == 1 && lastBtn == 0)
    {
        counter++;
    }

    //Wert des Tasters am Ende der Scheife zurückspeichern
    lastBtn = btn;
}
```

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

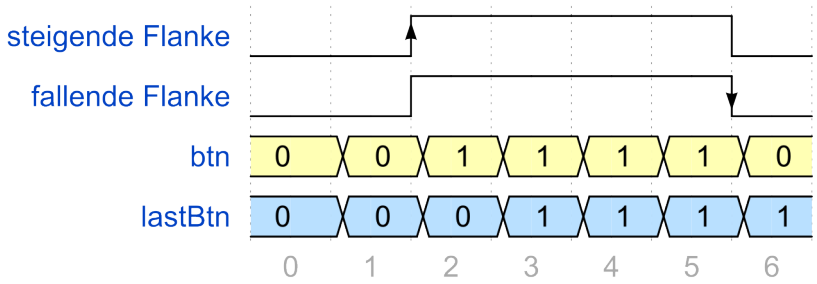

Abbildung 6: Werte der Variablen btn und lastBtn

Bei einer steigende Flanke hat lastBtn den Wert 0 und btn den Wert 1. Bei einer fallenden Flanke hat lastBtn den Wert 1 und btn den Wert 0.

Das Programm zum Zählen der Tastendrücke sieht dann wie folgt aus. Wobei in counterRising die steigenden Flanken und in counterFalling die fallenden Flanken gezählt werden.

```
int counterRising = 0;
int counterFalling = 0;
//Varible vor dem while(1) loop deklarieren
int lastBtn = 0;
while(1)
{
    int btn = PIND >> PIN_BTN & 0x01;
    if(btn == 1 && lastBtn == 0)
    {
        //steigende Flanke
        counterRising++;
    }
```

```
    if(btn == 0 && lastBtn == 1)
    {
        //fallende Flanke
        counterFalling++;
    }

    //Wert des Tasters am Ende der Scheife zurückspeichern
    lastBtn = btn;
}
```

Zeitsteuerungen

Für zeitliche Ablaufsteuerungen kann im einfachsten Fall die _delay_ms() Funktion verwendet werden.

```
while(1)
{
    PORTA |= 0x01 << 1;
    _delay_ms(500);
    PORTA &= ~(0x01 << 1);
    _delay_ms(500);
}
```

Diese Funktion realisiert ein Blinklicht mit 1hz Blinkfrequenz. Die _delay_ms() Funktion hat aber einen großen Nachteil. Sie blockiert jede weitere Funktion während Sie läuft. Ein abschaltbares Blinklicht würde z.B. so aussehen.

```
bool isBlinking = false;

while(1)
{
    bool btnBlinkOn = PINB >> 0 & 0x01;
    bool btnBlinkOff = PINB >> 1 & 0x01;

    if(btnBlinkOn)
    {
        isBlinking = true;
    }

    if(btnBlinkOff)
    {
        isBlinking = false;
    }

    if(isBlinking)
    {
        PORTA |= 0x01 << 1;
        _delay_ms(500);
        PORTA &= ~(0x01 << 1);
```

```
        _delay_ms(500);
    }
}
```

Wird dabei der Taster btnBlinkOff während einer _delay_ms Funktion gedrückt wird sich das Licht nicht ausschalten. Der Button muss während der Ausführung der ersten Zeilen gedrückt werden.

Eine nicht blockierende Zeitsteuerung kann mit der millis() Funktion programmiert werden. Dazu müssen die Dateien SystemClock.c und SystemClock.h im Projekt mitkompliert werden. Diese verwendet den internen Timer 0 zur Zeitmessung. Verwendet wird die SystemClock folgendermaßen:

```
#include "SystemClock.h"

int main()
{
    // Timer wird gestartet und beginnt im Hintergrund die
    // Zeit in millisekunden zu zählen.
    initTimer0AsSystemClock(true);

    //Verzweigung soll alle 500ms aufgerufen werden.
    unsigned long interval = 500;
    //Die Zeit des letzen Aufrufes muss gespeichert werden.
    unsigned long previousMillis = 0;

    while(1)
    {
        unsigned long currentMillis = millis();
        if(currentMillis >= previousMillis + interval)
        {
            //every 500ms
            previousMillis = currentMillis;
            LEDGREENTOGGLE;
        }
    }
}
```

Ein abschaltbares Blinklicht kann dann z.B. auf folgende Weise implementiert werden.

```
bool isBlinking = false;

unsigned long interval = 500;
unsigned long previousMillis = 0;  // will store last time LED was updated

while(1)
{
    unsigned long currentMillis = millis();

    bool btnBlinkOn = PINB >> PIN_BTN1 & 0x01;
    bool btnBlinkOff = PINB >> PIN_BTN2 & 0x01;
```

```
    if(btnBlinkOn)
    {
        isBlinking = true;
    }

    if(btnBlinkOff)
    {
        isBlinking = false;
    }

    if(isBlinking)
    {
        if(currentMillis >= previousMillis + interval)
        {
            //every 500ms
            previousMillis = currentMillis;
            LEDGREENTOGGLE;
        }
    }
    else
    {
        LEDGREENOFF;
    }
}
```