1 Ampelkreuzung

1.1 Ampelkreuzung: Michael Wörner (30137), Christian Silfang (30147)

1.2 Aufgabenstellung und -ziel

In der Aufgabe *Ampelkreuzung* soll ein komplexes Ampelsystem entwickelt werden. Hierfür steht eine Platform mit folgenden Bauteilen zur verfügung:

- 28 LEDs
- 8 Taster
- 4 Schieberegister seriell in parallel out vom Typ 74HCT595 und
- 2 Schieberegister parallel in seriell out vom Typ 74HCT165

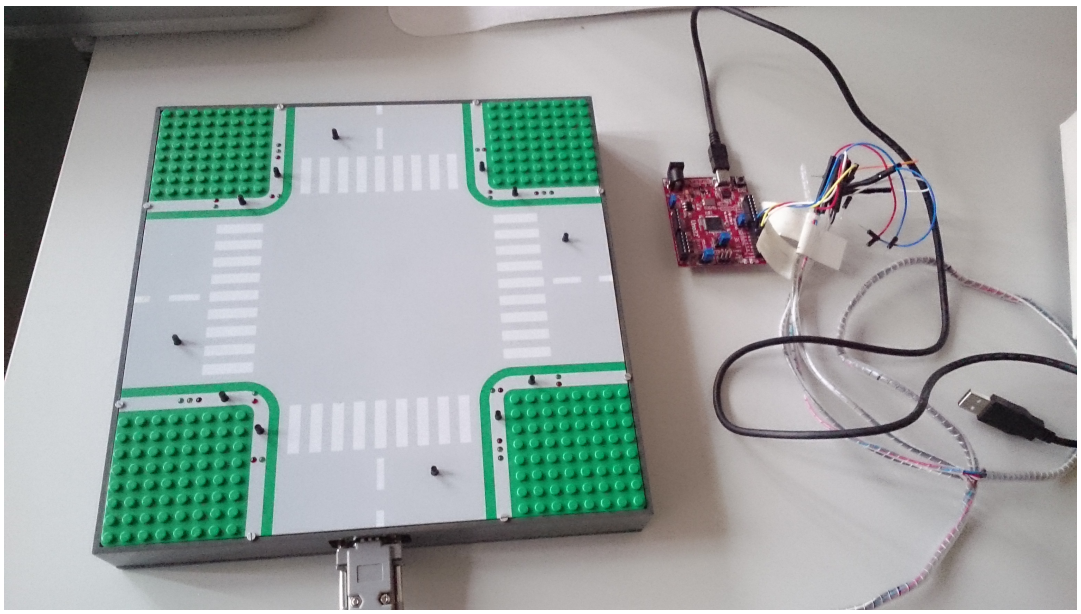

Abbildung 1.1: „Beweisfoto“ des Versuchs Ampelsteuerung

1.3 Versuchsaufbau

In diesem Abschnitt wird auf den Aufbau der Ampelsteuerung eingegangen und alle Kontextspezifischen Begriffe definiert.

Die Kreuzung besteht aus einer **Hauptstraße** (HS) und einer **Querstraße** (QS). Diese Begrifflichkeit dient rein zur Unterscheidung.

An der Kreuzung existieren Ampeln für **Fahrzeuge** und **Fußgänger**. Die Fußgängerampeln haben zwei Leuchten - rot und grün - und werden **Zweifeld-Ampel** genannt.

Ein **rot-Signal** bedeutet, dass der **Fußgängerübergang gesperrt** ist. Ein **grün-Signal** bedeutet, dass der Fußgängerübergang **frei** ist.

Die Ampeln für Fahrzeuge haben drei Leuchten - rot, gelb und grün. Diese werden **Dreifeld-Ampel** genannt. Ein rot-Signal bedeutet, dass die Straße *gesperrt* ist. Ein **rot-gelb-Signal** bedeutet, dass die Straße *noch gesperrt* ist. Ein grün-Signal bedeutet,

dass die Straße *frei* ist. Ein **gelb-Signal** bedeutet, dass die Straße *noch frei* ist und geräumt werden muss.

Die Zeitspannen in der ein Weg freigegeben ist wird *Grünphase* genannt.

Für die Erkennung eines Fahrzeugs ist im Modell ein Taster vorgesehen. Dieser soll eine **Induktionsschleife** (IS) simulieren.

Die Abb. 1.2 veranschaulicht den Aufbau der Ampelkreuzung.

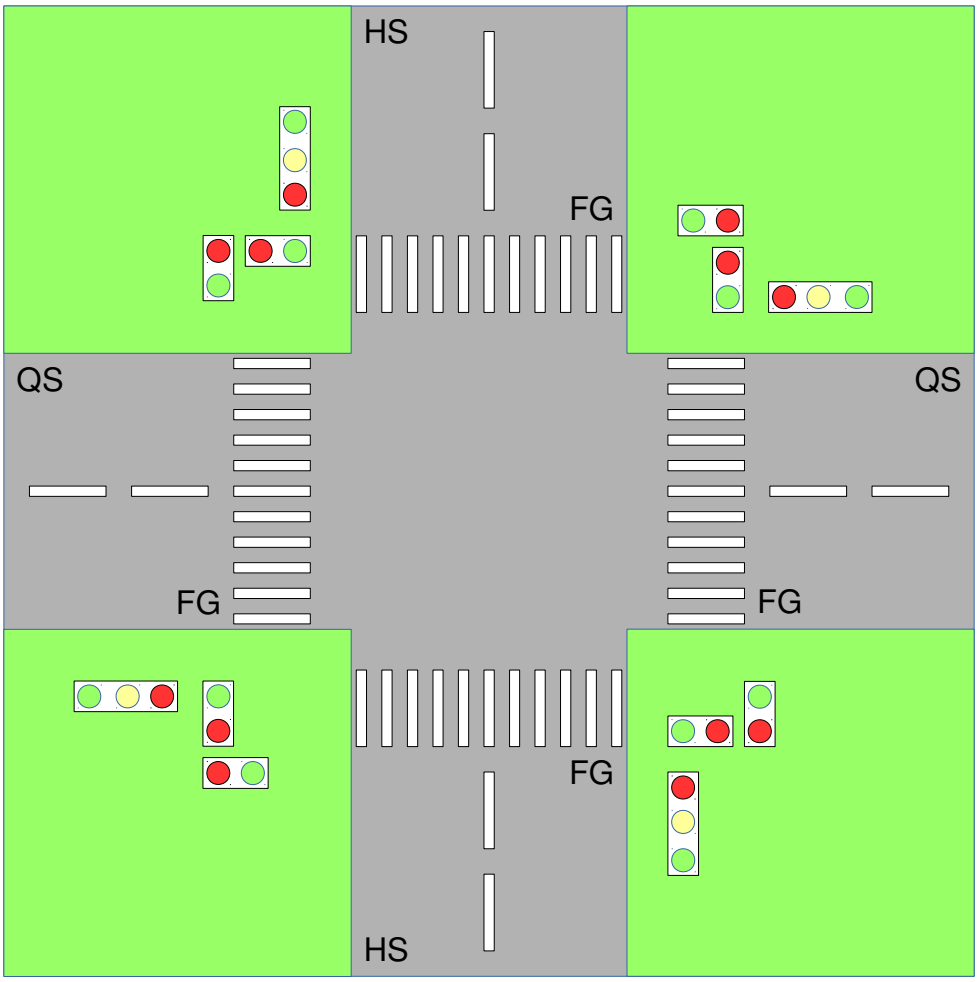

Abbildung 1.2: Aufbau der Ampelkreuzung

1.4 Modellierung

1.4.1 Definition der Ampelfunktionen und Zustände

Die Ampel soll den Verkehrsfulß einer Kreuzung regeln. Im folgenden ist aufgelistet wovon, dass sperren bzw. das freigeben der HS, QS uns FG abhängig ist.

- Die HS bzw. QS darf nur freigegeben werden wenn die QS bzw. HS gesperrt ist.
- Die FG darf nur freigegeben werden wenn die HS und die QS gesperrt ist.
- Die HS bzw. QS kann nur freigegeben werden, wenn in der vorherigen Grünphase diese nicht freigegeben wurde.

Die Ampel besitzt verschiedene Zustände. Deren Namen und Bedeutung ist in Tabelle 1.1 definiert

Zustandsname	**Zustands Erklärung**
ar	Alle Ampeln sind rot.
hs	Die Haupstraße ist freigegeben
qs	Die Querstraße ist freigegeben
fg	Der Fußgängerweg ist freigegeben

Tabelle 1.1: Definition der Zustände der Ampel

In Abb.1.3 ist ein UML Diagramm des Kompletten Zustandsautomaten dargestellt. In diesem sind alle Zustände und Übergänge sowie die Übergangsbedingungen abgebildet.

Dieser Zustandsautomat ermöglicht es die Haupt- und die Querstraße durch zwei Arten von Ereignissen frei zu geben:

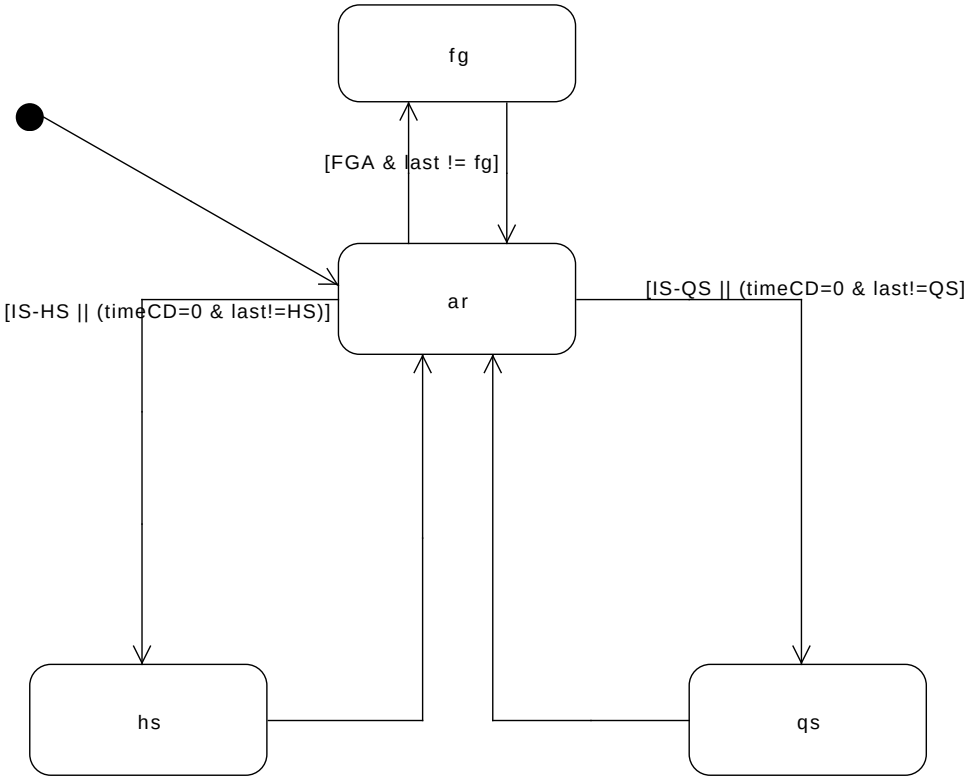

Abbildung 1.3: Die Zustände der Ampel in einem UML Diagramm

- Ablauf einer gewissen Zeit
- Reagieren auf ein ankommendes Fahrzeug mittels der Induktionsschleifen

Des weiteren wird auf das Betätigen eines Tasters an einer Fußgägngerampel reagiert. Bevor ein Zustandswechsel stattfindet, wird überprüft welcher Teil der Ampel als letztes frei gegeben wurde. Dies dient einmal dazu, dass die Haupt- und die Querstraße abwechselnd, in einem gewissen Zeitraum, freigegeben werden. Weiter noch verhindert dies, dass eine der Straßen oder der Fußgängerübergang blockiert wird.

1.5 Implementierung

In diesem Abschnitt soll nun näher auf die Realisierung der Ampelsteuerung eingegangen werden.

1.5.1 Setup des Boards

Die unten stehenden Tabellen zeigen die Belegungen der Pins für die Schieberegister von LEDs und Taster auf dem ChipKit Uno32. Dabei zeigt Tabelle 1.2 die Belegung der Pins für die LEDs und Tabelle 1.3 die Belegung der Pins für die benötigten Taster.

DIN1 (Serielle Daten IN)	34
SCK1 (Serieller Tackt)	35
CLK1 (Muss HIGH sein)	36
RCK1 (Serielles Update)	37

Tabelle 1.2: Pinbelegung der Schieberegister für die LEDs

DOUT2 (Serielle Daten OUT)	5
PL2 (Parallel Load)	6
CLK2 (Serieller Takt)	7

Tabelle 1.3: Pinbelegung der Schieberegister für die Taster

Die Implementierung des Setups ist in Listing 1.1 zu sehen.

```
1 void setup(void)
2 {
3     /* Initalisierung */
4     init_central_data();
5     init_shiftRegister();
```

```
6
7       /* Alle Ampeln auf Rot Schalten */
8       central_data.ampel_alle = MASK_ALL_RED;
9       writeLed();
10
11      /* Task der alle 50 ms Taster liest */
12      createTask(readSwitch, 50, TASK_ENABLE, &task_var);
13  }
```

Listing 1.1: Setup Funktion

Nachdem der Zentrale Datenspeicher sowie die Schieberegister und die LEDs initialisiert wurden, wird mittels der Funktion **createTask()** einen Task erstellt der alle 50ms Ausgeführt wird. Als Callbackfunktion wird die Funktion **readSwitch()** verwendet. Dadurch werden zyklisch die Taster ausgelesen. Die gelesenen Werte werden im zentralen Datenspeicher abgelegt. Wird ein Taster betätigt wird das dementsprechende Bit in der Variable **taster** gesetzt. Dieses wird erst wieder gelöscht wenn die dazugehörige Straßenseite freigegeben wurde.

1.5.2 Implementierung eines zentralen Datenspeichers

```
1  /* Definition des zentrale Datenspeichers */
2  typedef struct
3  {
4    uint32_t ampel_alle;
5      uint16_t taster;
6      uint8_t sr_out[4];
7  }central_data_t;
8
9  extern central_data_t central_data;
10
11 void init_central_data(void);
12
13 /* Implementierung des zentralen Datenspeichers */
14
15 central_data_t central_data;
16
17 void init_central_data(void)
18 {
19     uint8_t i;
20
21     central_data.ampel_alle = 0;
22     central_data.taster = 0;
23     for(i=0; i<4; i++)
24         central_data.sr_out[i] = 0;
25 }
```

Listing 1.2: Zentraler Datenspeicher

1.5.3 Festlegung der Bit-Masken für Ampelsignalfarben und zur Detektion der Taster

```
1  /* Fussgaenger gesperrt */
2  #define MASK_FG_RED          0b00010100000101000001010000010100
3  /* Fussgaenger frei*/
4  #define MASK_FG_GREEN        0b00001010000010100000101000001010
5
6  /* Hauptstrasse gesperrt*/
7  #define MASK_HS_RED          0b00000000100000000000000010000000
8  /* Hauptstrasse frei*/
9  #define MASK_HS_GREEN        0b00000000001000000000000000100000
10 /* Hauptstrasse noch gesperrt*/
11 #define MASK_HS_YELLOWRED    0b00000000110000000000000011000000
12 /* Hauptstrasse noch frei*/
13 #define MASK_HS_YELLOW       0b00000000010000000000000001000000
14
15 /* Querstrasse gesperrt*/
16 #define MASK_QS_RED          0b10000000000000001000000000000000
17 /* Querstrasse frei */
18 #define MASK_QS_GREEN        0b00100000000000000010000000000000
19 /* Querstrasse noch gesperrt*/
20 #define MASK_QS_YELLOWRED    0b11000000000000001100000000000000
21 /* Querstrasse noch frei*/
22 #define MASK_QS_YELLOW       0b01000000000000000100000000000000
23
24 /* Alle Wege sind geperrt */
25 #define MASK_ALL_RED         0b10010100100101001001010010010100
26
27 /* Maskierung der Taster Fussgaenger */
28 #define FGA     0b0011011000110110
```

```
29 /* Maskierung der Taster Induktionsschleife Querstrasse */
30 #define IS_QS  0b0000100000001000
31 /* Maskierung der Taster Induktionsschleife Hauptstrasse */
32 #define IS_HS  0b0100000001000000
```

Listing 1.3: Bit-Masken zur Festlegung der Ampelsignalfarben und zur Detektion der Taster

1.5.4 Steuerung der Kreuzung über einen Zustandsautomaten

```
1  void loop(void)
2  {
3    static street_t aktState = ar, lastState = qs;
4    static unsigned int cntTime = 0;
5
6    /* Zustandsautomat */
7    switch(aktState)
8    {
9      case ar:
10         /* Zustand: Alle Rot */
11         aktState = alleRot( (millis() - cntTime), lastState );
12     break;
13
14     case hs:
15         /* Zustand: Freigabe der Hauptstrasse */
16         aktState = dreiFeldAmpel(hs);
17         lastState = hs;      /* hs als letzten Zustand abspeichern
               */
18         cntTime = millis(); /* Zeit festhalten */
19     break;
20
21     case qs:
```

```
22          /* Zustand: Freigabe der Querstrasse */
23          aktState = dreiFeldAmpel(qs);
24          lastState = qs;      /* qs als letzten Zustand abspeichern
                */
25          cntTime = millis(); /* Zeit festhalten */
26      break;
27
28      case fg:
29          /* Zustand: Freigabe des Fussgaengeruebergangs */
30        aktState = zweiFeldAmpel();
31        cntTime = millis();    /* Zeit festhalten */
32        lastState = fg;        /* fg als letzten Zustand abspeichern
              */
33      break;
34
35      default:
36      break;
37    }
38
39  }
```

Listing 1.4: Zustandsautomat zur Ampelsteuerung

Um die einzelnen Ampeln zu schalten bzw. um auf ein Drücken der Taster zu agieren wurde als Hauptsteuerung der Ampelkreuzung ein Zustandsautomat implementiert der in Listing 1.4 dargestellt ist. Dabei beziehen sich die einzelnen Zustände auf die, die bereits in Abschnitt 1.4.1 definiert worden sind.

Um diesen Zustandsautomaten zu implementieren werden drei Funktionen benötigt - **alleRot()**, **dreiFeldAmpel()** und **zweiFeldAmpel()**. Die genaue Implementierung dieser Funktionen ist in Abschnitt 1.5.5 zu sehen. Der Rückgabe wert dieser Funktionen

ist der nächste zustand der Ampel.

Der Zustandsautomat wird Zyklisch in der main-loop ausgeführt und besteht aus einer Switch-Case Anweisung. Die einzelnen Cases sind die Zustände, die in der Variable **aktState** abgespeichert werden.

Im case **ar** wird die Funktion **alleRot** ausgeführt. Dieser Funktion wird die Differenz aus der aktuellen Zeit und dem Zeitpunkt an dem sich die letzte Zustandsänderung ereignet hat übergeben. Des weiteren wird der Zustand vor der letzten Zustandsänderung übergeben.

Im case **hs** wird die Funktion **dreiFeldAmpel()** aufgerufen. Dieser wird übergeben, dass sich die Ampel gerade im Zustand **hs** befindet. Diese Funktion steuert die Dreifeldampeln - in diesem Fall die der Hauptstraße. In der Variable **lastState** wird der Zustand **hs** abgespeichert. Des weiteren wird die Aktuelle Zeit in der Variable **cntTime** festgehalten.

Im case **qs** wird die gleiche Prozedur wie in **hs** durchgeführt. Es wird lediglich der Funktion **dreiFeldAmpel()** übergeben, dass nun die Querstraße fregegeben ist.

Für die Implementierung des Zustandes **fg** existiert der case **fg**. In diesem wird die Funktion **zweiFeldAmpel()** aufgerufen. Zusätzlich wird, genau wie in **hs** und **qs** der aktuelle Zustand und die aktuelle Zeit gesichert.

1.5.5 Steuerung der Ampelsignalfarben

Da auf dieser Kreuzung Zwei- und Drei-Feld-Ampeln existieren mussten zwei verschiedene Funktionen zur Ansteuerung des Signals implementiert werden: Das Listing 1.5 zeigt den Code für eine Zwei-Feld-Ampel, das Listing 1.6 für Drei-Feld-Ampeln.

```
1 /* Funktion fuer die Steuerung der Zweifeldampeln */
2 /* Der Rueckgabewert der Funktion ist der naechste Zustand der
```

```
   Ampel */
street_t zweiFeldAmpel(void)
{
    static color_t aktColor = red;
    static uint32_t cntTime = 0;
    uint32_t dT;
    street_t s_return;

    /* Berechnung der Zeitdifferenz seit dem letzten aufruf */
    dT = millis() - cntTime;
    if( aktColor == red )
    {
        /* Ampel von rot auf gruen schalten */
        central_data.ampel_alle &= ~MASK_FG_RED;
        central_data.ampel_alle |= MASK_FG_GREEN;
        writeLed();
        cntTime = millis();

        /* Naechster Zustand ist "Fussgaenger" */
        aktColor = green;
        s_return = fg;
    }
    else if( (aktColor == green) && ( dT > GREEN_TIME) )
    {
        /* ampeln von gruen auf rot schalten */
        central_data.ampel_alle &= ~MASK_FG_GREEN;
        central_data.ampel_alle |= MASK_FG_RED;
        writeLed();

        /* Naechster Zustand ist "alleRot" */
        aktColor = red;
```

```
33          s_return = ar;
34
35          /* Taster-Werte loeschen */
36          central_data.taster &= ~FGA;
37      }
38      else s_return = fg; /* Zustand ist Weiterhin Fussgaenger */
39
40      return s_return;
41  }
```

Listing 1.5: Funktion zur Steuerung der Ampelsignale einer Zwei-Feld-Ampel

Die Aufgabe der Funktion **zweiFeldAmpel()** ist es den Zeitlichen Ablauf der Fußgänger Amplen zu steuern. Der Zeitliche Ablauf dieser Funktion sieht folgendermaßen aus:

1. Nach dem ersten aufrufen dieser Funktion werden alle Signale der Fußgängerampel auf grün umgeschaltete. Dies geschieht indem die jeweiligen Bits im Zentralen Datenspeicher geändert werden und anschließend die Funktion **writeLed()** aufgerufen wird. Anschließend wird in einer Statischen Variable die Aktuelle Zeit abgespeichert.

   In diesem Fall gibt die Funktion den Wert **fg** zurück, um zu Signalisieren das die Ampel sich weiterhin im Zustand **fg** befindet.

2. Bei jedem weiteren Aufrufen dieser Funktion, wird überprüft ob die Zeit, seit dem die Fußgängerampeln auf grün sind, größer ist als die Definierte Grünphase. Ist dies der Fall werden alle Fußgängerampeln wieder auf rot zurückgestellt. Anschließend gibt die Funktion den Wert **ar** zurück um zu Signalisieren, dass sich die Ampel wieder im Ausgangszustand **ar** befindet.

   Für den Fall, dass die Zeit der Grünphase noch nicht abgelaufen ist, gibt die

Funktion erneut den Wert **fg** zurück um weiterhin in diesem zustand zu bleiben.

```
/* Funktion fuer die Steuerung der Dreifeldampeln */
/* Der Funktion muss uebergeben werde ob die Haupt- oder die
   Querstrasse gesteuert wird */
/* Die Funktion gibt den naechsten Zustand der Ampel zurueck */
street_t dreiFeldAmpel(street_t street)
{
    /* Variablendeklaration */
    static color_t aktColor = red;
    static uint32_t cntTime = 0;
    uint32_t dT;
    street_t s_return;

    /* Zeitdifferenz berechnen */
    dT = millis() - cntTime;

    if(aktColor == red)
    {
      /* Ampel von rot auf rot-gelb schalten */
      if(street == hs)
      {
        central_data.ampel_alle &= ~MASK_HS_RED;
        central_data.ampel_alle |= MASK_HS_YELLOWRED;
        s_return = hs;
      }
      else
      {
        central_data.ampel_alle &= ~MASK_QS_RED;
        central_data.ampel_alle |= MASK_QS_YELLOWRED;
        s_return = qs;
      }
```

```
30      writeLed();
31      cntTime = millis();
32      aktColor = yellowred; /* Naechste Farbe ist gelbrot */
33    }
34    else if( (aktColor ==  yellowred) && ( dT > YELLOW_TIME) )
35    {
36        /* von rot-gelb auf gruen schalten */
37        if(street == hs)
38        {
39            central_data.ampel_alle &= ~MASK_HS_YELLOWRED;
40            central_data.ampel_alle |= MASK_HS_GREEN;
41            s_return = hs;
42        }
43        else
44        {
45            central_data.ampel_alle &= ~MASK_QS_YELLOWRED;
46            central_data.ampel_alle |= MASK_QS_GREEN;
47            s_return = qs;
48        }
49        writeLed();
50        cntTime = millis();
51        aktColor = green;
52    }
53    else if( (aktColor ==  green) && (dT > GREEN_TIME) )
54    {
55        /* von gruen auf gelb schalten */
56        if(street == hs)
57        {
58            central_data.ampel_alle &= ~MASK_HS_GREEN;
59            central_data.ampel_alle |= MASK_HS_YELLOW;
60            s_return = hs;
```

```
61          }
62          else
63          {
64              central_data.ampel_alle &= ~MASK_QS_GREEN;
65              central_data.ampel_alle |= MASK_QS_YELLOW;
66              s_return = qs;
67          }
68          writeLed();
69          cntTime = millis();
70          aktColor = yellow;
71      }
72      else if((aktColor ==  yellow) && (dT > YELLOW_TIME))
73      {
74          /* von gelb auf rot schalten */
75          if(street == hs)
76          {
77              central_data.ampel_alle &= ~MASK_HS_YELLOW;
78              central_data.ampel_alle |= MASK_HS_RED;
79              central_data.taster &= ~ IS_HS;
80          }
81          else
82          {
83              central_data.ampel_alle &= ~MASK_QS_YELLOW;
84              central_data.ampel_alle |= MASK_QS_RED;
85              central_data.taster &= ~ IS_QS;
86          }
87          writeLed();
88          cntTime = millis();
89
90          aktColor = red;
91          s_return = ar;
```

```
92      }
93      else
94      {
95          if(street == hs) s_return = hs;
96          else s_return = qs;
97      }
98
99      return s_return;
100 }
```

Listing 1.6: Funktion zur Steuerung der Ampelsignale einer Drei-Feld-Ampel

Die Funktion **dreiFeldAmpel()** erledigt im Grunde die gleiche Funktionalität wie die Funktion **zweiFeldAmpel()**. Damit diese den Ablauf einer Dreifeldampel übernehmen kann muss diese mit den folgenden Funktionalitäten erweitert werden:

- Unterscheidung ob die Haupt- oder die Querstraße freigeben werden soll
- zeitliche Steuerung der Gelbphase und die
- zeitliche Steuerung der Rotgelbphase.

Für die Unterscheidung ob die Haupt- oder die Querstraße freigegeben wird, muss der Funktion dieser Parameter beim aufrufen übergeben werden. Des weiteren ist in dieser Funktion noch eine zusätzliche Zeitabfrage für die Gelb- bzw. Rotgelphase implementiert. Das freigeben einer Straße hat dabei folgenden Zeitlichen Ablauf

1. Wechsel von rot auf rot-gelb. Dies geschieht unmittelbar nach dem ersten aufrufen der Funktion. Dieser Zustand hält so lange an bis eine Definierte Zeit abgelaufen ist.
2. Anschließend erfolgt ein Wechsel von rot-gelb nach grün.

3. Nachdem die Zeit der Grünphase abgelaufen ist werden die Ampeln, der entsprechenden Straße, auf gelb geschallten.

4. Nachdem auch die Zeit der Gelbphase abgelaufen ist werden die Ampeln wieder auf rot geschallten und die Ampel wechselt wieder in den Zustand **ar**.

Zusätzlich wurde zur Initialisierung und zum Wechsel der Ampelsignale eine Funktion implementiert, die alle Signale zunächst auf *rot* setzt um einen sicheren Zustand der gesamten Kreuzung zu gewährleisten. Die Funktion ist in Listing 1.7 zu sehen.

```
1  /* Funktion fuer die Bestimmung des naechsten */
2  /* Zustandes der Ampel */
3  /* Der Funktion muss die Zeitdifferenz seit der */
4  /* Letzten freigabe einer Strasse uebergeben werden */
5  /* Der Funktion muss die zuletzt freigebene Strasse */
6  /*ueberbegeben werden um eine Blockierun zu verhindern */
7  /* Die Funktion gibt den naechsten Zustand der Ampel zurueck */
8  street_t alleRot(unsigned int dT, street_t lastStreet)
9  {
10     street_t s_return;
11
12     if ( (central_data.taster & FGA) && (lastStreet != fg) )
          s_return = fg;
13     else if( (central_data.taster & IS_HS) && (lastStreet != hs)
          ) s_return = hs;
14     else if( (central_data.taster & IS_QS) && (lastStreet != qs)
          ) s_return = qs;
15     else if( dT > RED_TIME )
16     {
17         if( lastStreet != hs ) s_return = hs;
18         else s_return = qs;
19     }
```

```
20      else s_return = ar; /* Ampel bleibt rot */
21
22      return s_return;
23  }
```

Listing 1.7: Funktion zur Steuerung der Ampelsignale: Sicherheitszustand (alle rot)

Die Funktion **alleRot()** steuert den Ablauf während sich die Ampel im Zustand **ar** befindet. Dabei muss sie folgende aufgaben übernehmen:

- auswerten ob ein Taster der Fußgängerübergänge betätigt wurde
- auswerten ob eine Induktionsschleife der Haupt- oder der Querstraße betätigt wurde und
- überprüfen wie lange sich die Ampel in der Rotphase befindet.

Um dies durchzuführen muss der Funktion die vergangenen Zeit, seit der letzten Freigabe der Haupt- oder Querstraße bzw des Fußgängerübergangs, sowie die Information welcher Straßenteil als letztes freigegeben wurde übergeben werden.

Abhängig davon welches Ereignis (ablaufen einer definierten Zeit oder die Betätigung eines Tasters) statt gefunden hat, gibt die Funktion den nächsten Zustand der Ampel zurück.

Da das Abfragen der Taster hintereinander geschieht ist es möglich, dass die Reaktion auf ein Ereignis die Reaktion auf ein anderes Ereignis blockieren kann. Aus diesem Grund wird beim Abfragen, ob ein Taster betätigt wurde, gleichzeitig noch abgefragt, ob auf dieses Ereignis bereits reagiert wurde.

1.5.6 Ansteuerung der Hardware

Für die Implementierung der Ampelsteuerung werden Funktion für das Schreiben und Lesen der Schieberegister benötigt. Da ein Lese- bzw. Schreibzyklus Byteorientiert geschieht, werden auch Funktionen für die Konvertierung in die dementsprechende Datenformate benötigt.

```
1  /* Funktion um die Schieberegister zu initalisieren */
2  /* Hierbei werden die dementsprechenden Pins eingestellt */
3  void init_shiftRegister(void)
4  {
5      /* Schieberegister out */
6      pinMode(DIN1, OUTPUT);
7      pinMode(SCK1, OUTPUT);
8      pinMode(CLK1, OUTPUT);
9      pinMode(RCK1, OUTPUT);
10
11     digitalWrite(DIN1, LOW);
12     digitalWrite(SCK1, LOW);
13     digitalWrite(RCK1, LOW);
14     digitalWrite(CLK1, HIGH);
15
16     /* Schieberegister in */
17     pinMode(DOUT2, INPUT);
18     pinMode(PL2, OUTPUT);
19     pinMode(CLK2, OUTPUT);
20
21     digitalWrite(PL2, LOW);
22     digitalWrite(CLK2, LOW);
23
24 }
```

Listing 1.8: Funktion zur Initialisierung der Schieberegister

Für die LEDs sind 4 Schieberegister mit einer Bitbreit von 8 Bit vorgesehen. Die Register sollen immer an einem Stück beschrieben werden. Damit aber eine einzige 32 Bit Variable verwendet werden kann wurden zwei Funktionen für die Konvertierung implementiert.

```
1  /* Funktion um eine long-Variable in vier Char-Variablen
       umzuwandeln */
2  /* Der Funktion wird eine Vorzeichenlose long-Variable */
3  /* sowie ein Zeiger auf char-Array*/
4  /* Das Char-Array muss mindestens 4 Elemente enthalten */
5  void long2char(uint32_t in, uint8_t *out)
6  {
7      uint32_t temp;
8
9      temp = in;
10     out[0] = (uint8_t) (temp & 0x000000ff);
11
12     temp = in;
13     out[1] = (uint8_t) ( (temp >> 8) & 0x000000ff);
14
15     temp = in;
16     out[2] = (uint8_t) ( (temp >> 16) & 0x000000ff);
17
18     temp = in;
19     out[3] = (uint8_t) ( (temp >> 24) & 0x000000ff);
20 }
21
22
23 /* Funktion um ein Char-Array  mit 4 Elementen in eine*/
24 /* Vorzeichenlose Long-Variable umzuwandeln */
```

```
25 /* Der Funktion wird ein Zeiger auf ein Array mit mindestens */
26 /* 4 Elementen uebergeben */
27 /* Die Funktion gibt die Umgewandelte Long-Variable zurueck */
28 uint32_t char2long(uint8_t *in)
29 {
30    uint32_t u32_return = 0;
31    uint8_t i;
32 
33    for(i=0; i<4; i++)
34    {
35       u32_return |= ( (long) (in[i] << i));
36    }
37 
38    return u32_return;
39 }
```

Listing 1.9: Funktionen für die Konvertierung einer long Variable in 4 char Varibalen und vice versa

Aufgrund eines Seltsamen verhalten der *MPIDE* musste die Arduino-Funktion **shiftIn()** manuell in den Sourcecode mitaufgenommen werden. Diese Funktion dient dazu eine 8 Bit Variable in ein Schieberegister zu schreiben. Der Funktion wird übergeben an welchem Pin sich der Datenausgang befindet, an welchem Pin sich der Takt befindet und in welcher Reihenfolge die Daten übertragen werden sollen.

```
1 /* Diese Funktion wurde aus der Arduino Umgebung heraus kopiert
     */
2 /* Der Funktion muss uebergeben werden an welchen Pin sich der */
3 /* der Datenausgang befindet, an welchem Pin sich der Taktausgang
      */
4 /* befindet und in welcher reihenfolge die Bits rausgeschoben
     werden */
```

```
5  uint8_t shiftIn(uint8_t dataPin, uint8_t clockPin, uint8_t
       bitOrder) {
6    uint8_t value = 0;
7    uint8_t i;
8
9    for (i = 0; i < 8; ++i) {
10     digitalWrite(clockPin, HIGH);
11     if (bitOrder == LSBFIRST)
12       value |= digitalRead(dataPin) << i;
13     else
14       value |= digitalRead(dataPin) << (7 - i);
15     digitalWrite(clockPin, LOW);
16   }
17   return value;
18 }
```

Listing 1.10: Funktion zum schreiben eines Bytes in eine Schieberegister

Damit alle Schieberegister auf einmal beschrieben werden, wurde die Funktion **writeLed()** implementiert. In dieser Funktion wird als erstes die 32 Bit Variable **ampel_alle** aus dem zentralen Datenspeicher in vier 8 Bit Variablen umgewandelt. Anschließend werden diese vier Bytes mit der Arduino-Funktion **shiftOut()** in die Schieberegister übertragen. Wehrend Der Übertragung muss ein Steueranshluss am Schieberegister auf 0V gehalten werden.

```
1  /* Diese Funktion um die LEDs zu beschreiben */
2  /* Die Daten - welche LEDs aktiviert bzw deaktiviert werden -
       werden aus dem Zentralen */
3  /* Datenspeicher entnommen */
4  void writeLed(void)
5  {
6      uint8_t i, u8_data[4];
```

```
7
8       /* 32 Bit Variable in vier 8 Bit Variablen umwandel */
9       long2char(central_data.ampel_alle, u8_data);
10
11      digitalWrite(RCK1, LOW);
12      for(i=0; i<4; i++)
13      {
14          shiftOut(DIN1, SCK1, LSBFIRST, u8_data[i]);
15      }
16      digitalWrite(RCK1, HIGH);
17  }
```

Listing 1.11: Funktionen zum Schreiben der LEDs

Zuletzt wird noch eine Funktion benötigt um die Taster über die Schieberegister auszulesen. Die Funktion **readSwitch()** wird als callback-Funktion in einem Task Verwendet. Hierbei wird die Arduino-Funktion **shiftIn()** dazu verwendet, die zwei Schieberegister auszulesen. Die zwei gelesenen Bytes werden in einer 16 Bit Variable abgespeichert. Bevor das Schieberegister gelesen werden kann muss ein Steueranschluss auf 3,3V gehalten werden. Bei einer steigenden Flanke an diesem Pin werden die Werte am Eingang des Schieberegister in das interne Register übernommen.

```
1  /* Funktion fuer das auslesen der Taster */
2  /* Diese Funktion wird zyklisch in einem Task alle 50ms
       aufgerufen */
3  /* Die Funktion gibt nichts zurueck */
4  /* Der Funktion muss ein integer und ein Zeiger auf eine integer-
       */
5  /* Variabel uebergeben werden. Dies ist eine Vorgabe der Funktion
       */
6  /* createTask() */
7  void readSwitch(int a, void *b)
```

```
8  {
9      uint8_t u8_data[2];
10     uint16_t i, temp = 0;
11
12
13     digitalWrite(PL2, HIGH);     /* Paralell Input load */
14     for(i=0; i<2; i++)
15         u8_data[i] = shiftIn(DOUT2, CLK2, MSBFIRST);
16
17     digitalWrite(PL2, LOW);
18
19     /* Daten in Zentralen Datenspeicher schreiben */
20     central_data.taster |= (uint16_t) (u8_data[1] << 8);
21     central_data.taster |= (uint16_t) u8_data[0];
22 }
```

Listing 1.12: Funktionen zum Lesen der Taster

1.6 Aufteilung des Sourcecodes

Da der Sourcecode etwas umfangreich ausfällt, wird er in mehrere Dateien ausgelagert. Jede dieser .cpp-Dateien besitzt eine eigene Header-Datei. In den Header Dateien befinden sich die Einbindungen aller notwendigen Bibliotheken sowie die Funktionsprototypen aller Öffentlichen Funktionen.

Die Dateien sind folgendermaßen aufgeteilt

- ampelsteuerung.pde
  - Diese Datei beinhaltet die setup- und loop-Funktion.

- data.h und data.cpp
  - In der Datei data.h ist der zentrale Datenspeicher definiert. In der Datei data.cpp befindet sich die Funktion **init_central_data()**.
- shift.cpp
  - In der Datei shift.cpp befinden sich die Implementierungen der Funktionen für die Ansteuerung der Schieberegister sowie die Funktionen für die Konvertierung der Variablen
- traficLight.cpp
  - In dieser Datei befinden sich die Funktionen für die Steuerungen der Ampelsignalen
- def.h
  - In der Datei def.h befinden sich alle Definitionen. Dies beinhaltet die Anschlüsse, die Zuordnung der LEDs zu den Bits in einer 32 Bit Variable, die Zuordnung der Taster zu den Bits in einer 16 Bit Variable sowie die Zeiten für die Rot-, Rot-Gelb, Gelb- und Grünphase.