Da der tatsächliche Stromverbrauch der Bauteile deutlich unterhalb der 2A liegt, wird die Leiterbahnbreite ohne große Sicherheit auf 0.51mm festgelegt.

6. Software

Aufbauend auf dem Schaltplan und dem Layout, welche definieren wie die Signale den Mikrocontroller erreichen, soll nun die Verarbeitung dieser Signale in der Software betrachtet werden. Dabei soll für alle für alle Komponenten zuerst um den Aufbau und den Ablauf der Software gehen, ohne die tatsächliche Software zu betrachten. Anschließend sollen diese Abläufe in Programmcode umgesetzt und näher betrachtet werden erläutert werden. Aufgabe der Software ist es die Signale, welche dem Mikrocontroller zur Verfügung gestellt werden, gemäß den Anforderungen, welche in früheren Kapiteln an das System gestellt wurden, zu verarbeiten. Dieser Ablauf soll mittels Flussdiagramme betrachtet werden, wie es in Abbildung 22 beispielhaft dargestellt ist. Dabei geht es darum Abläufe schematisch und übersichtlich darzustellen. Es werden große Abläufe in einzelne kleine Prozessschritte unterteilt und deren zeitlicher und logischer Ablauf grafisch dargestellt. Die Abläufe sind dabei von Funktionen und Variablen losgelöst, dass ist erst Aufgabe der Software den Ablauf, welcher hier definiert wird, tatsächlich in funktionierenden Programmcode umzusetzen. Diese Ablaufdiagramme sollen nun für alle wichtige Abläufe erstellt und erklärt werden. Diese Betrachtung soll dabei für alle Komponenten und Kommunikations-abläufe betrachtet werden. Anschließend soll die Umsetzung dieser Abläufe in Software erklärt werden.

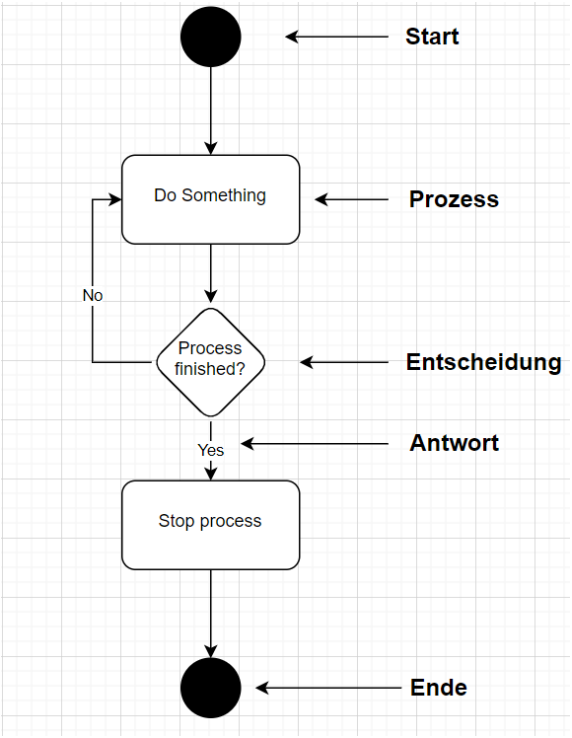

Abbildung 22: Beispielhafter Ablauf eines Flussdiagramms

Dabei gibt es einige Grundlegende Voraussetzungen, welchen der gesamte Code folgen soll. Im Vordergrund soll die schnelle Verarbeitung aller Signale stehen. Es soll daher vermieden werden, die Eingangspins von Signalen via Polling, also das zyklische Auslesen des Eingangsregisters, abzufragen. Stattdessen sollen die Eingänge einen Interrupt auslösen, sodass die Abfrage eines Signals nur durchgeführt wird, wenn tatsächlich eine Veränderung am Pin erkannt wurde. Dabei ist darauf zu achten, die Interrupt Service Routinen (ISR) so knapp wie möglich zu halten, da diese die Abarbeitung des restlichen Codes blockieren. Vor allem auf Delays, Wartefunktionen oder Kommunikation mit anderen Bauteilen muss in der ISR verzichtet werden. Stattdessen setzt jede ISR lediglich

ein Flag, welches im normalen Programmcode abgearbeitet werden kann. Diese Abarbeitung geschieht in der Loop Funktion. Das ist die Funktion, welche als Dauerschleife vom Mikrocontroller ausgeführt wird. Hier werden alle Prozesse abgearbeitet, wenn das jeweilige Flag gesetzt wurde. Ist kein Flag gesetzt, findet keine Abarbeitung statt. Damit die Abarbeitung alle Funktionen zügig geht, ohne dass sich Abläufe gegenseitig blockieren, soll im gesamten Code auf Delays durch aktives Warten verzichtet werden. Es soll möglich sein, dass während einzelne Funktionen blockiert sind, andere weiterhin abgearbeitet werden können. Neben der Loop Funktion gibt es die Funktion Setup(). Diese wird einmal zu Beginn des Programmablaufs ausgeführt und soll alle Initialisierungen und Konfigurationen der Bauteile und Pins vornehmen. Zu diesen im Setup ausgeführten Funktionen gehört die Funktion init_ports(), welche immer zu Beginn eines jeden Programmstarts abläuft.

```
322  // Init Ports
323  void init_ports(){
324
325    Serial.println("Init Ports");
326    // Output
327    pinMode(SPI_CS_CAN2_PIN, PIN_OUTPUT);
328    pinMode(SPI_CS_DISPLAY_PIN, PIN_OUTPUT);
329    pinMode(SPI_CS_FLASH_PIN, PIN_OUTPUT);
330    pinMode(SPI_RST_LCD_PIN, PIN_OUTPUT);
331    pinMode(SPI_RST_TP_PIN, PIN_OUTPUT);
332
333    // Input
334    pinMode(SPI_INT_CAN2_PIN, PIN_INPUT);
335    pinMode(SPI_INT_TP_PIN, PIN_INPUT);
336    pinMode(INT_PE_PIN, PIN_INPUT);
337
338    // UART
339    MicroUSB.begin(115200, SERIAL_8N1, UART_RX_PIN, UART_TX_PIN);
340
341    // I2C
342    Wire.begin(I2C_SDA_PIN, I2C_SCL_PIN);
343
344    // SPI
345    my_SPI.begin(SPI_CLK_PIN, SPI_MISO_PIN, SPI_MOSI_PIN);
346
347    //PWM For Display Backlight
348    analogWrite(LCD_BL_PIN, 140);
349  }
```

Programmcode 1: Funktion init_ports()

Diese legt die Funktionalität aller Pins als Input, Output oder Alternate Funktion zur Verwendung für Busse oder Interrupts fest. Zur besseren Lesbarkeit wurden Defines als Pinnamen definiert, hinter welchen sich der tatsächliche Pin verbirgt. Die Funktion PinMode legt dabei einen Pin als klassischen GPIO-Input oder Output fest zur

Verwendung als klassischer Digitaler Pin. Die Pins für SPI, I^2C und UART- Kommunikation werden über die Funktionen begin() festgelegt, welche in den jeweiligen Librarys definiert sind. So bezeichnet Wire das Objekt, über welches die I^2C Kommunikation gesteuert wird. Über die Funktion analogWrite() wird der Pin für das PWM-Signal für die Hintergrundbeleuchtung des Displays definiert und konfiguriert. Der übergebene Parameter regelt dabei den Duty Cycle, also das Verhältnis zwischen High und Low. Der Parameter wird dabei als 8-Bit Wert übergeben, wobei 255 einem Duty Cycle von 100% entspräche [36, S. 396]. Bei dem übergebenen Pin beträgt der berechnete, optimierte Duty Cycle:

$$D = \frac{140}{255} = 55\%$$

Die Interrupts werden zur Übersichtlichkeit in einer eigenen Funktion definiert, welche ebenfalls im Setup aufgerufen wird. Dabei handelt es sich um die 3 Pins, welche bereits als Input definiert wurden.

```
427  void init_Interrupts(){
428
429    Serial.println("Init Interrupts");
430    // Init Interrupts
431    // Interrupt GPIO Expansion
432    attachInterrupt(digitalPinToInterrupt(INT_PE_PIN),
433      ISR_GPIO_Expansion, RISING);
434
435    // Interrupt CAN2-Controller
436    attachInterrupt(digitalPinToInterrupt(SPI_INT_CAN2_PIN),
437      ISR_CAN2, RISING);
438
439    // Interrupt Touch Display
440    attachInterrupt(digitalPinToInterrupt(SPI_INT_TP_PIN),
441      ISR_TouchController, RISING);
442  }
```

Programmcode 2: Funktion init_Interrupts()

Diese werden mit der Funktion attachInterrupt als Interruptquelle definiert. Zusätzlich wird die Interrupt Service Routine übergeben, die Funktion, welche ausgeführt wird, sobald ein Interrupt eintritt.

Die Signale werden jedoch nicht alle direkt vom Mikrocontroller selbst eingelesen. Einige werden über die Porterweiterung erfasst und per I^2C vom Mikrocontroller abgefragt. Auch diese Pins müssen dafür entsprechend konfiguriert werden, was ebenfalls per I^2C Kommunikation vom ESP32 übernommen wird.

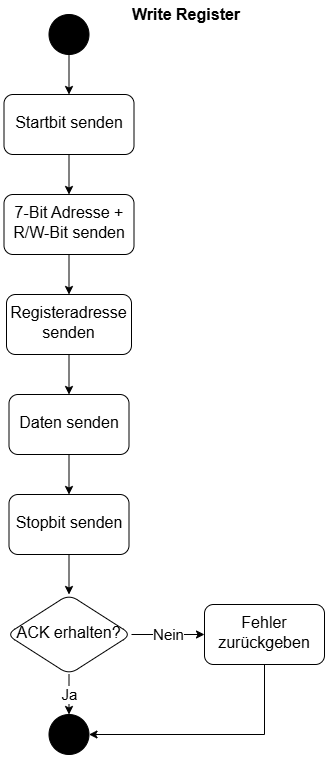

Abbildung 23: Ablaufdiagramm für I²C-Kommunikation

Die Form der Kommunikation gibt das Datenblatt vor und wird in Abbildung 23 als Flussdiagramm dargestellt [21, S. 14ff]. Das Bauteil erwartet ein Startbit mit seiner Adresse und dem Read/Write-Bit. Das ist die Adressierung, um zu bestimmen welcher Chip angesprochen werden soll. Der Chip antwortet daraufhin mit einem Acknowledge-Bit. Wird dieses Bit nicht gesendet, bedeutet das, dass die Adressierung nicht erfolgreich war, entweder durch eine Falsche Adresse oder Fehlgeschlagene Kommunikation. In diesem Fall soll ein Fehler zurückgegeben werden und die Funktion beendet werden. War die Adressierung erfolgreich, wird als nächstes die Adresse des zu beschreibenden Registers übermittelt. Auch hierauf antwortet der Chip mit einem Acknowledge. Bleibt dieses aus, zum Beispiel wegen einer falschen Registeradresse, soll auch hier wieder ein Fehler zurückgegeben und die Funktion beendet werden. Im Code wird diese Funktionalität in der Funktion GPIO_Exp_WriteRegister() umgesetzt.

In dieser Funktion werden zuerst die an die übergebenen Parameter überprüft. Dabei handelt es sich um die Registeradresse und die Daten. Die Daten dürfen dabei die Größe eines Bytes nicht überschreiben. Die Registeradressen müssen sich in einem definierten Bereich zwischen 0x00 – 0x0A oder 0x10 -0x1A befinden [21, S. 12]. Hier soll die Abarbeitung der Funktion beendet werden, bevor ein Fehler während der Kommunikation auftreten kann.

```
359
360  int GPIO_Exp_WriteRegister(int reg, int value){
361
362    // Check for Value out of Range
363    if(value > 255 || value < 0){
364      // Value out of Range
365      Serial.println("Value out of Range");
366      return ERROR;
367    }
368
369    if(reg<0x00 || (reg < 0x10 && reg > 0x0A) || reg > 0x1A){
370      Serial.println("Register Address out of Range");
371      return ERROR;
372    }
```

Programmcode 3: Überprüfung der Eingaben zum Beschreiben eines Registers der GPIO-Porterweiterung

```
373
374     // Write Address and Write Bit to Buffer
375     // I2C Adresse + Write Bit
376     Wire.beginTransmission(GPIO_EXP_ADRESS << 1);
377
378     // Write register Address to Buffer
379     int rv = Wire.write(reg);
380     if(rv <= 0){
381       Serial.println("Write register Adress failed");
382       return ERROR;
383     }
384
385     // Write Data to Buffer
386     rv = Wire.write(value);      // Write Data
387     if(rv <= 0){
388       Serial.println("Write data failed");
389       return ERROR;
390     }
391
392     // Send Buffer to Communication
393     rv = Wire.endTransmission();
394     if(rv != 0){
395       Serial.println("Sending Bufferr to Communication failed");
396       return ERROR;
397     }
398     return SUCCESS;
399   }
```

Programmcode 4: I²C-Kommunikation zum Beschreiben eines Registers der GPIO-Porterweiterung

Um die Kommunikation aufzubauen, wird zuerst mit der Funktion beginTransmission() die Adresse geschrieben. Die Funktion fügt dabei automatisch das Startbit an die Adresse hinzu. Die geschriebenen Adressen und Daten werden dabei nicht direkt übermittelt, sondern zuerst in einen Buffer geschrieben. Dasselbe gilt für die Funktion write(). Hier werden zuerst die Registeradresse und anschließend die Daten in den Buffer geschrieben. Die Funktion gibt dabei zurück, wie viele Bytes erfolgreich in den Buffer geschrieben wurden. Ein Rückgabewert von 0 bedeutet entsprechend, dass keine Daten in den Buffer geschrieben wurden. In diesem Fall wird die Funktion unter Rückgabe eines Fehlers beendet. Die Serielle Ausgabe dient dabei zum Debugging, während man per USB mit dem System verbunden ist, um so genauere Fehlerbeschreibungen zu erhalten. Mit der Funktion endTransmission() wird der Inhalt des Buffers nun gesendet. Wird ein gerät unter der angegebenen Adresse gefunden, antwortet dieses mit einem Acknowledge. Der Empfang des Acknowledge-Bit wird vom Sendenden Chip empfangen. Die erfolgreiche Kommunikation wird durch die Rückgabe der Funktion endTransmission() erkannt. Ist dieser Wert 0, ist kein Fehler aufgetreten. Neben der Funktion GPIO_Exp_WriteRegister() gibt es auch die Funktion GPIO_Exp_WriteBit().

```
468
469  int GPIO_Exp_WriteBit(int reg, int bit, int value){
470
471    // Check for value in Range
472    if(value > 1 || value < 0){···
476    else{···
479
480    // Check for Address out of Range
481    if(reg<0x00 || (reg < 0x10 && reg > 0x0A) || reg > 0x1A){···
485    else{···
488
489    int rv = 0;
490    // Read Data from Register
491    int reg_value = GPIO_Exp_ReadRegister(reg);
492    if(reg_value == ERROR){
493      Serial.println("Register could not be read");
494      return ERROR;
495    }
496
497    // Changes Bit in Register Data and write data to Buffer
498    if(value == HIGH){
499      rv = GPIO_Exp_WriteRegister(reg, reg_value | (1 << bit));
500    }
501    else if(value == LOW){
502      rv = GPIO_Exp_WriteRegister(reg, reg_value & ~(1 << bit));
503    }
504
505    // Error handling
506    if(rv == ERROR){
507      Serial.println("Writing Failed");
508      return ERROR;
509    }
510
511    return SUCCESS;
512  }
```

Programmcode 5: I²C-Kommunikation zum Schreiben eines Bits in Registern der GPIO-Porterweiterung

Diese Funktion funktioniert genau wie die Funktion WriteRegister(), mit dem Unterschied, dass sie nur ein Bit im adressierten Register verändert. Da es nur möglich ist das gesamte Register zu adressieren, liest die Funktion als erstes das entsprechende Register und übergibt diesen Wert mit dem einen modifizierten Bit an die Funktion WriteRegister(), welche das Register neu beschreibt. Dafür wird zusätzlich zu den Schreibfunktionen für die Porterweiterung auch eine Lesefunktion benötigt, welche die adressierten Register ausliest und zurückgibt. Anders als bei der Schreibfunktion wird schon nach dem Schreiben der Registeradresse gesendet und auf ein Acknowledge-Bit gewartet. Hier wird jedoch anders als bei dem Schreibbefehl noch kein Stoppbit gesendet. Wird das ACK-Bit erhalten, wird der Wert aus dem Register gelesen und von der Funktion zurückgegeben. Im Code wird der Anfang identisch zur Schreibfunktion umgesetzt.

```
400
401   int GPIO_Exp_ReadRegister(int reg){
402
403     // Check for Address out of Range
404 >   if(reg<0x00 || (reg < 0x10 && reg > 0x0A) || reg > 0x1A){···
408
409     // Write Address and Write Bit to Buffer
410     // Device Adress + Readbit
411     Wire.beginTransmission((GPIO_EXP_ADRESS << 1) || READ);
412
413     // Write register address to Buffer
414     int rv = Wire.write(reg);
415     if(rv <= 0){ return ERROR;}
416
417     // Send Buffer to Communication
418     rv = Wire.endTransmission(false);  // Repeated Start, No Stoppbit
419     if(rv != 0){ return ERROR;}
420
421     // Write Address and Read bit to Buffer
422     // Send Request for Register value (1 Byte)
423     Wire.requestFrom(GPIO_EXP_ADRESS, 1);
424
425     // Read Register from Communication
426     if (Wire.available()) {
427       // Read from Receive Buffer
428       return Wire.read(); // Return Byte
429     }
430     return ERROR; // Receive Buffer empty
431   }
```

Programmcode 6: I²C-Kommunikation zum Lesen eines Registers der GPIO-Porterweiterung

Als erstes wird geprüft, dass sich die Registeradresse in einem gültigen Adressbereich befindet. Der erste Unterschied lieg darin, dass statt einer 0 eine 1 an die Adresse als R/W-Bit angehängt wird, um eine Leseoperation zu kennzeichnen. Im Anschluss wird auch hier die Registeradresse in den Buffer geschrieben. Schlägt eine dieser Schreibfunktionen fehl, wird die Funktion mit einer Fehlerrückgabe beendet. Anders als bei der Schreibfunktion werden hier keine Daten mehr geschrieben, sondern es folgt endTransmission zum Senden den Buffers. Das ist nötig, damit der Chip für die folgende Leseanfrage weiß, für welches Register diese gilt. Daher ist auch darauf zu achten mit dem Senden der Daten kein Stoppbit zu schicken, da die Kommunikation ohne gelesene Daten noch nicht abgeschlossen ist. Das wird durch den Parameter „false“ erreicht, welcher der Funktion endTransmission() übergeben wird. Auch hier wird wieder auf eine korrekte Übertragung und ein erhaltenes Acknowledge-Bit geprüft. Über die Funktion requestFrom() wird nun der Lesevorgang eingeleitet. Mit dieser Funktion stellt der Mikrocontroller eine Leseanfrage an die Porterweiterung. Dabei werden wieder die Geräteadresse und die Anzahl an erforderten Bytes übergeben. Auf diese Anfrage antwortet der Chip mit dem Inhalt des Registers. Dieser wird im Eingangsbuffer

gespeichert und kann dort mit Wire.read() ausgelesen werden. Vorher wird mittels der Funktion available() abgefragt, ob sich neue Daten im Buffer befinden. Sollte diese Abfrage fehlschlagen und der ESP32 keine neuen Daten erhalten haben, wird ein Fehler zurückgegeben, andernfalls beendet sich die Funktion mit der Rückgabe der erhaltenen Daten.

```
477
478  int GPIO_Exp_ReadBit(int reg, int bit){
479
480    // Check for Bit Value out of Range
481    if(bit > 7){ return ERROR;}
482
483    // Check for Address out of Range
484 >  if(reg<0x00 || (reg < 0x10 && reg > 0x0A) || reg > 0x1A){···
488
489    // Read Register from GPIO Port Expansion
490    int value = GPIO_Exp_ReadRegister(reg);
491    if(value == ERROR){ return ERROR;}
492
493    // mask and filter for single bit
494    u_int8_t bit_value = value >> bit;
495    bit_value &= 0x01;
496
497    return bit_value;
498  }
```

Programmcode 7: I²C-Kommunikation zum Lesen eines Bits aus den Registern der GPIO-Porterweiterung

Ähnlich wie bei den Schreibfunktionen gibt es auch hier eine Funktion, welche nur ein Bit zurückgibt. Das es nur möglich ist ganze Register auszulesen, nutzt die Funktion GPIO_Exp_ReadBit() die ReadRegister() Funktion und maskiert am Ende den erhaltenen Wert, um einen Bitwert zurückgeben zu können. Die Funktionen werden nun auch genutzt, um das Bauteil entsprechend der Anforderungen zu konfigurieren. Dazu gehört die Konfigurierung der Pins als Input/Output, sowie anderer Register. Dies findet wie die Konfigurierung der Mikrocontroller Pins in der Setup-Funktion statt. Dort wird die Funktion init_GPIO_Expansion() aufgerufen. In dieser Funktion wird als erstes das IO-Control Register beschrieben. In diesem Register kann die Adressierung der Register definiert werden. Durch das Setzen von Bit7 des Registers werden die Adressen für Bank A und Bank B blockweise unterteilt, sodass die Adressen 00-0A für die Register von GPIOA genutzt werden. So können nun die entsprechenden Register korrekt adressiert werden. Als nächstes sollen die Ports als Ein und Ausgänge definiert werden. Dafür müssen die Register IO_Direction A (IODIRA) und IO-Direction B (IODIRB) beschrieben werden [21, S. 18].

```
350
351  int init_GPIO_Exp_Ports(){
352
353    // SET Bank for addressing
354    int rv = GPIO_Exp_WriteBit(0x05, 7, HIGH);
355  >   if(rv == ERROR){ ···
359
360    {
361    rv = 0;
362    // set Pin Direction
363    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRA, 0, PIN_INPUT); // LED Learn
364    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRA, 1, PIN_INPUT); // LED Signa
365    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRA, 2, PIN_INPUT); // RC Errors
366    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRA, 3, PIN_INPUT); // RC_Reciev
367    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRA, 5, PIN_INPUT); // RC Receiv
368    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRA, 7, PIN_INPUT); // RC Receiv
369    if(rv != 6){Serial.println("Error in init Ports Bank A"); return ERR
370    else{Serial.println("Init Ports Bank A successful");}
371
372    rv = 0;
373    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 0, PIN_OUTPUT); // LED Pin
374    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 1, PIN_OUTPUT); // Activate
375    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 2, PIN_OUTPUT); // RFID SPI
376    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 3, PIN_OUTPUT); // RFID SPI
377    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 4, PIN_OUTPUT); // RC Trans
378    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 5, PIN_OUTPUT); // RC Trans
379    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 6, PIN_OUTPUT); // CAN1 Sil
380    rv += GPIO_Exp_WriteBit(GPIO_EXP_IODIRB, 7, PIN_OUTPUT); // CAN2 Sil
381    if(rv != 8){Serial.println("Error in init Ports Bank B"); return ERR
382    else{Serial.println("Init Ports Bank B successful");}
383    }
384  >   { ···
391  >   { ···
416  >   { ···
422    return SUCCESS;
423  }
```

Programmcode 8: Initialisierung der GPIO-Porterweiterung

Pin 0 entspricht dabei Bit 0. Die Ports werden nun entsprechend dem Schaltplan definiert. Jede Aufgerufene Funktion gibt dabei ihren Rückgabewert aus, welcher bei erfolgreicher Bearbeitung 1 betragen muss. Damit nicht jeder Funktionsaufruf einzeln kontrolliert werden muss, werden die Rückgabewerte summiert und blockweise überprüft. Um manche Signale korrekt auswerten zu können, bietet das Bauteil die Option aus der Software heraus Pull-Up Widerstände zu aktivieren. Diese werden hier nicht benötigt, da alle Pull-Up Widerstände bereits im Schaltplan zur Bestückung vorgesehen sind. Um keine Signale zu verfälschen, werden die Pull-Up Widerstände für alle Pins deaktiviert. Als letztes müssen die Interruptquellen definiert werden. Dafür werden zuerst alle

Eingangspins als Interruptquelle freigegeben. Das passiert über das Register GP_Interrupt_Enable_A (GPINTENA) [21, S. 19]. Hier werden alle benötigten Bits auf 1 gesetzt. Das wird über den Wert 0xAF für die Bits 7, 5, 3, 2, 1 und 0 erreicht. Zur Auslosung von Interrupts gibt es 2 Varianten. Entweder wird ein Interrupt ausgelöst, wenn der Wert dem Invertierten Wert entspricht, welcher im Default Value Register (DEFVALA) definiert ist oder der Chip löst einen Interrupt bei jeder Änderung der Pin-Levels aus. Das Interruptverhalten wird dabei im Interrupt Control Register (INTCONA) festgelegt. Soll der Wert gegen das DEFVAL-Register vergleicht werden, muss das entsprechende Bit im INTCONA auf 1 gesetzt werden [21, S. 20]. Das ist der Fall für die Pins 5, 3, 2 und 0. Für diese Pins muss entsprechend auch der Wert im DEFVALA-Register gesetzt werden. Bei den Signalen an Pin 0 und 2 handelt es sich um lowaktive Signale, weshalb der Wert hierfür auf High gesetzt werden muss, da der Interrupt immer beim invertierten Signalpegel ausgelöst wird. Die Werte für Pins 3 und 5 werden auf Low gesetzt.

```
351   int init_GPIO_Exp_Ports(){
360   {···
384   {···
391   {
392   // set Interrupt Settings
393   rv = 0;
394   // Enable Interrupts for Pins 7, 5, 3, 2, 1, 0
395   rv += GPIO_Exp_WriteRegister(GPIO_EXP_GPINTENA, 0xAF);
396
397   // Define Comparison Values for Pins to throw Interrupts
398   // Interrupt is set, if opposite value occurd
399   rv += GPIO_Exp_WriteBit(GPIO_EXP_DEFVALA, 0, HIGH); // low-active Si
400   rv += GPIO_Exp_WriteBit(GPIO_EXP_DEFVALA, 2, HIGH); // low-active Si
401
402   rv += GPIO_Exp_WriteBit(GPIO_EXP_DEFVALA, 3, LOW);
403   rv += GPIO_Exp_WriteBit(GPIO_EXP_DEFVALA, 5, LOW);
404
405   // Set Interrupt Control Regsiter to comparison against previous
406   //      Values or DEFVAL Regsister
407
408   // Pins 5, 3, 2 and 0 are compared against defVal (set to 1)
409   // Pins 7 and 1 are compared against previous Value (set to 0)
410   // -> 0b 0010 1101
411   rv += GPIO_Exp_WriteRegister(GPIO_EXP_INTCONA, 0x2D);
412
413   // Set Interrupt polarity to High_active
414   rv += GPIO_Exp_WriteBit(GPIO_EXP_IOCONA, 1, HIGH);
415
416   if(rv != 7){return ERROR;}
417   }
418   {···
424   return SUCCESS;
425   }
```

Programmcode 9: Interrupt Einstellungen für die GPIOO-Porterweiterung

Die Initialisierungen der Interrupts ergeben sich durch die Funktionalitäten der an den Pins angeschlossenen Signale. Die Pins 0 und 2 lesen die beiden lowaktiven LEDs vom Funkmodul ein und sollen die Impulse zählen, welche zur Fehler- oder Modusidentifizierung ausgesendet werden. Dafür sollen sie auf jeden Low-Pegel reagieren. Ähnliches gilt für Pins 3 und 5. Hier liegen die Signale an, welche das Funkmodul für den Pegel der Kanäle 2 und 3 verwendet. Diese sind Highaktiv, weshalb der Mikrocontroller bei jedem Highpegel über das Eintreffen eines Funksignals informiert werden soll. Anders sieht es bei den Signalen 1 und 7 aus. Diese schalten die Status-LED. Das bedeutet, das bei jedem High-pegel die LED eingeschalten werden soll, bei jedem Low-Pegel aber auch wieder aus. Daher wird hier immer gegen den vorherigen Wert am Pin verglichen und nicht gegen einen fest definierten wert. Zuletzt soll über IOCAN1 noch die Polarität des Interrupt-Signals an den ESP32 gesetzt werden, welches hier als Highaktiv definiert wird.
Zuletzt wird noch einer der Ausgänge beschalten.

```
351   int init_GPIO_Exp_Ports(){
418     {
419     // Initial Value
420     rv = 0;
421     rv += GPIO_Exp_WriteBit(GPIO_EXP_GPIOB, 1, HIGH);
422     if(rv != 1){return ERROR;}
423     }
424     return SUCCESS;
425   }
```

Programmcode 10: Setzen der Initialen Werte für Pins der GPIO-Porterweiterung

Dort ist an Pin1 ein lowaktives Signal angeschlossen, welches standardmäßig nicht aktiv sein soll. Dieses wird daher auf High gezogen. Dafür wird das GPIOB-Register passend beschrieben.
Wird ein Interrupt von der GPIO-Erweiterung an den Mikrocontroller gesendet, reagiert der Chip auf die Steigende Flanke und führt die Interrupt Service Routine aus. Das ist für die Porterweiterung die Funktion ISR_GPIO_Expansion().

```
740
741   // Interrupts
742   void ISR_GPIO_Expansion(){
743     // Interrupt Service Routine for GPIO Expansion Interrupts
744     ISR_GPIO_Exp_Flag = true;
745   }
```

Programmcode 11: Erkennen eines Interrupts durch die GPIO-Porterweiterung

Diese Funktion macht nichts anderes als ein Flag zu setzen, welches dafür sorgt, dass der Interrupt im nächsten durchlauf des Loop abgearbeitet wird. Die Abarbeitung des Interrupts erfolgt in der Funktion process_ISR_GPIO_Expansion().

In dieser wird als erstes das Interrupt_Flag-Register (INTFA) ausgelesen. In diesem Register ist das Bit aktiv, für welches ein Interrupt generiert wurde. Dieses wird nun maskiert und für jedes Bit gecheckt und entsprechend verarbeitet. Die Verarbeitung soll für die jeweiligen Signale gesondert betrachtet werden. Dabei wird das Register durch das Auslesen Automatisch zurückgesetzt [21, S. 22].

```
747   void process_ISR_GPIO_Expansion(){
748     // Read Interrupt Pending Register
749     // Set Flags for Interrupts
750     int flags = GPIO_Exp_ReadRegister(GPIO_EXP_INTFA);
751     int rv = GPIO_Exp_ReadRegister(GPIO_EXP_INTCAPA);
752
753     if(flags != ERROR){
754
755       // Check for every single bit
756       if((flags >> 0) & 0x01){
757         // Interrupt ocured for Bit 0
758         // RF Controller Pairing Mode Acknowledge
759         ISR_Learn_LED_CTR += 1;
760       }
761 >     if((flags >> 1) & 0x01){···
766 >     if((flags >> 2) & 0x01){···
771       if((flags >> 3) & 0x01){
772         // Interrupt ocured for Bit 3
773         // Remote Drive Command received from RF Module
774         ISR_RX_2_Flag = true;
775       }
776 >     if((flags >> 4) & 0x01){···
780       if((flags >> 5) & 0x01){
781         // Interrupt ocured for Bit 5
782         // SOC Command received from RF Module
783         ISR_RX_3_Flag = true;
784       }
785 >     if((flags >> 6) & 0x01){···
789 >     if((flags >> 7) & 0x01){···
802
803     }
804   }
```

Programmcode 12: Verarbeitung der Interrupts der GPIO-Porterweiterung für jedes Signal

Die Betrachtung für die Verarbeitung des Interrupts soll anhand der Pins 3 und 5 durchgeführt werden. Dabei handelt es sich um die empfangenen Signale für Kanal 2 und 3 des Funkmoduls.

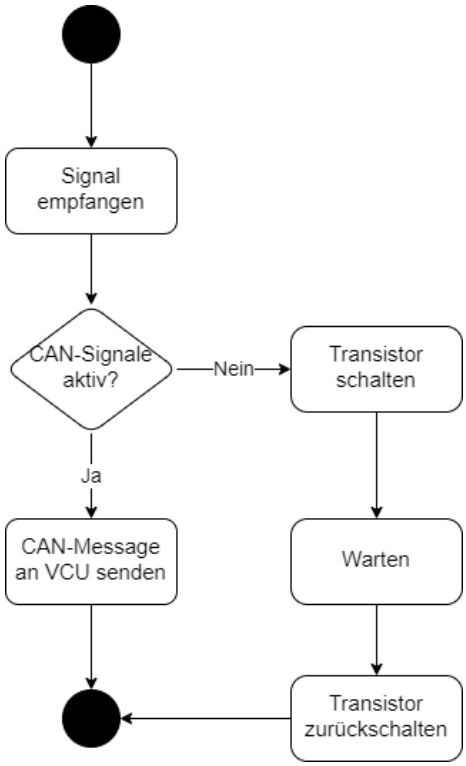

Abbildung 24: Ablaufdiagramm für Behandlung empfangener Signale vom Funkmodul

Nachdem das Signal am Mikrocontroller empfangen wurde, soll als erstes gecheckt werden, welche Einstellungen aktiv sind. Standardmäßig soll das Signal als CAN-Message auf CAN-Bus 1 an die VCU gesendet werden. Es gibt jedoch auch alte Systeme, welche noch nicht über CAN-Kommunikation verfügen. Für diese ist es möglich die Signale über einzelne Signalleitungen zu übertragen. In diesem Fall muss dafür der entsprechende Transistor geschalten werden, welcher das Signal erzeugt. Dieser muss nach einer kurzen Wartezeit auch wieder deaktiviert werden. Programmiertechnisch soll die Umsetzung beispielhaft an Kanal2 gezeigt werden. In der Interrupt Serviceroutine des GPIO-Expanders wird ein Flag ISR_RX_2 gesetzt, wenn der entsprechende Pin für das Funksignal als Interruptquelle identifiziert wurde. Dieses Flag wird im Loop zyklisch abgefragt und bei Bedarf die Funktion send_RemoteDrive_Request() aufgerufen.

```
1274  int send_RemoteDrive_Request(){
1275
1276    static bool Signal_active = false;
1277    static int timer_end = 0;
1278    int rv = 0;
1279
1280    if(ESP_storage.getInt("RF_CAN_en", FALSE) == FALSE){
1281      // Send analog Signal
1282      // Set Pin GPIOB 4 on Port Extension to HIGH to power the MOSFET
1283      if(Signal_active == false){
1284        Signal_active = true;
1285        timer_end = millis() + 300; // Set Timer for 300ms
1286
1287        int rv = GPIO_Exp_WriteBit(GPIO_EXP_GPIOB, 4, HIGH);
1288        if(rv == ERROR){
1289          Serial.println("Write failed");
1290          ISR_RX_2_Flag = false; // Reset Flag
1291          Signal_active = false;
1292          return ERROR;
1293        }
1294      }
1295      if(millis() < timer_end){
1296        return 0; // Wait until 300ms are over
1297      }
```

```
1298
1299      // Reset Flag
1300      ISR_RX_2_Flag = false;
1301      Signal_active = false;
1302
1303      // Set Pin GPIOB 4 on Port Extension to LOW to reset the MOSFET
1304      rv = GPIO_Exp_WriteBit(GPIO_EXP_GPIOB, 4, LOW);
1305      if(rv == ERROR){
1306        Serial.println("Write failed");
1307        return ERROR;
1308      }
1309    }
```

Programmcode 13: Erzeugung des Remote Drive Signals an das Steuergerät

Diese Funktion prüft als erstes, ob das Signal analog oder per CAN gesendet werden soll. Die Einstellungen dafür sind im Non-Volatile Storage (NVS) des ESP32 gespeichert, einem Speicherbereich, welcher auch nach Reboot erhalten bleibt. Wenn das Signal analog gesendet werden soll, wird nun der Transistor geschalten. Dieser wird über Pin 4 der Porterweiterung gesteuert, weswegen dort ein Bit über die bereits betrachteten Funktionen gesetzt wird. Zusätzlich wird der aktuelle Zeitpunkt zuzüglich einer Wartezeit gespeichert. Das Flag wird noch nicht zurückgesetzt. Das liegt an dem Delay, welches zwischen ein und Ausschalten abgewartet werden muss. Um nicht die Abarbeitung aller anderen Funktionen zu blockieren, wird hier nicht die Funktion delay() genutzt. Stattdessen wird die Funktion zyklisch aufgerufen. Die weitere Abarbeitung ist jedoch so lange unterbrochen, bis der aktuelle Zeitpunkt vor dem liegt, welcher zu beginn, zuzüglich der Wartezeit, gespeichert wurde. Auf diese Weise unterbricht die Funktion keine anderen Prozesse. Damit der Ausgang der Porterweiterung nicht bei jedem Durchlauf erneut beschrieben wird, wird der erste Teil der Funktion durch die Variable Signal_active verriegelt, welche erst bei einem Fehler oder nach Ablauf des Delays zurückgesetzt wird. Nach Ablauf dieser Wartefunktion wird der Transistor wieder zurückgesetzt, sodass an der VCU ein Signalimpuls erkannt wird. Zu diesem Zeitpunkt wird auch das Flag aus dem Loop zurückgesetzt, um die weitere Abarbeitung zu verhindern. Ist die Einstellung, die Signale via CAN zu schicken aktiv, so wird die Funktion send_CAN1_Message() mit entsprechenden Parametern aufgerufen. Diese Funktion soll später genauer betrachtet werden. Alle hier getätigten Aussagen gelten auch für empfangene Signale von Kanal 2, lediglich mit Änderung der Signalnamen.

Für empfangene Signale auf Kanal 4 gibt es jedoch einige Unterschiede. Hierbei handelt es sich um das Signal zur Identifizierung des Karts über die Statusleuchte. Schon bei der Abfrage der Interrupt Flags unterscheidet sich dieser Kanal von den anderen. Anders als bei Kanal 2 und 3 wird jedes Mal, wenn sich der Zustand des Pins ändert ein Interrupt gesendet. Das hängt damit zusammen, dass die LED so lange Leuchten soll, wie der Knopf auf der Fernbedienung gedrückt wird. Das Flag wird daher nur von der ISR des Port Expanders verändert.

```
747   void process_ISR_GPIO_Expansion(){
753     if(flags != ERROR){
780       if((flags >> 5) & 0x01){···
785       if((flags >> 6) & 0x01){···
789       if((flags >> 7) & 0x01){
790         // Interrupt ocured for Bit 7
791         // Status LED Command received from RF Module
792         if((rv >> 7) & 0x01){
793           // Value during interrupt is HIGH -> Button is pressed
794           // Status LED Switch on
795           ISR_RX_4_Flag = true;
796         }
797         else{
798           // Status LED Switch off
799           ISR_RX_4_Flag = false;
800         }
801       }
```

Programmcode 14: Behandlung des Interrupts für einen LED-Request vom Funkmodul

Die Auswertung wiederum findet wie bei den anderen Kanälen im Loop statt. Hier muss jedoch zwischen den Unterschiedlichen Stati der LED unterschieden werden. So soll die LED nur geschalten werden, wenn sie aktuell aus ist und das Flag gesetzt ist. Andersherum soll sie nur ausgeschalten werden, wenn das Flag nicht gesetzt ist, die LED aber noch leuchtet. Zu diesem Zweck wird der Zustand der LED in einer Variable ID_cur_state gespeichert. Dabei handelt es sich nicht um den realen Status der LED, sondern um einen pseudozustand, welcher jedes Mal verändert wird, wenn das Flag abgearbeitet wird. Dabei wird davon ausgegangen, dass die LED zu Beginn immer ausgeschalten ist. Über die Funktionen Status_LED_ON() und Status_LED_OFF() wird dann der Transistor, welcher als Treiber für die LED fungiert, geschalten oder zurückgesetzt. Dieser wird über Pin 0 an der Porterweiterung geschalten. Die LED wird jedoch nicht nur von dem Funkempfänger gesteuert, es muss auch möglich sein ihren Zustand von der VCU aus zu verändern. Hierbei gibt es zwei Varianten wie dieses Signal an die Erweiterungsplatine übermittelt werden soll. Für alte Signale wird das Signal über die bestehende Signalleitung im Kabelbaum übermittelt. Das Signal wird dann über die Porterweiterung ausgelesen. Dabei löst das Signal einen Interrupt aus. Genau wie Kanal 4 des Funkempfängers, wird bei jeder Veränderung am Pin ein Interrupt generiert. Dieser setzt und deaktiviert ein Flag, welches im Loop ausgewertet wird und dort die LED schaltet. Wichtig ist dabei, dass diese Abarbeitung des Flag nur stattfindet, wenn die Funktionalität der LED auch vorhanden ist. Auch hier wird darauf geachtet, dass die LED nur geschalten wird, wenn ihr aktueller Zustand das auch erlaubt. Dafür ist die variable LED_cur_state vorgesehen, welche ebenso wie die Variable ID_cur_state einen Pseudozustand der LED speichert. Zusätzlich wird auch der Status von ID_cur_state abgefragt, also ob aktuell ein Kart über die Fernbedienung identifiziert wird, da das Funksignal Priorität gegenüber dem VCU-Signal besitzt. Das hat den Hintergrund, dass die LED zu blinken beginnt, sobald das Kart in einen Ready-To-Drive Zustand versetzt wird.

Auch in diesem Modus soll es noch möglich sein das Kart über die Fernbedienung zu identifizieren.

```
226   if(ESP_storage.getInt("LED_enable", FALSE) == TRUE){
227     if(ISR_LED_Signal_Flag == true && LED_cur_state == OFF
228       && ID_cur_state == OFF){
229
230     // The Controller has recieved a Signal from VCU,
231     //    it has to activate Status LED
232     // The LED must be switched on as long as the signal is active
233     // This Signal low priority compared to ID via RF Control
234       LED_cur_state = ON;
235       Status_LED_ON();
236     }
237     if(ISR_LED_Signal_Flag == false && LED_cur_state == ON
238       && ID_cur_state == OFF){
239
240     // The Microcontroller has recieved a Signal from VCU,
241     //    Status LED activation has ended
242     // The LED must be switched off as the signal is no longer active
243     // This Signal low priority compared to ID via RF Control
244       LED_cur_state = OFF;
245       Status_LED_OFF();
246     }
247   }
```

Programmcode 15: Abfrage des LED-Flag und des aktuellen LED-Status zum Schalten der LED

Die Zweite Variante, wie der Mikrocontroller Information zur Status-LED erhält ist per CAN-Message. Diese Option soll zu einem Späteren Zeitpunkt betrachtet werden.

Damit der Funkempfänger Signale empfangen kann, muss zuerst eine Fernbedienung verbunden werden können. Dafür sendet der Mikrocontroller ein Signal an das Funkmodul, welches daraufhin in einen Pairing-Modus eintritt. Dabei gibt es verschiedene Lernmodi, in welche das Modul versetzt werden kann. Die Abläufe zur Aktivierung dieser Modi sind als Ablaufdiagramm in Abbildung 25 dargestellt.

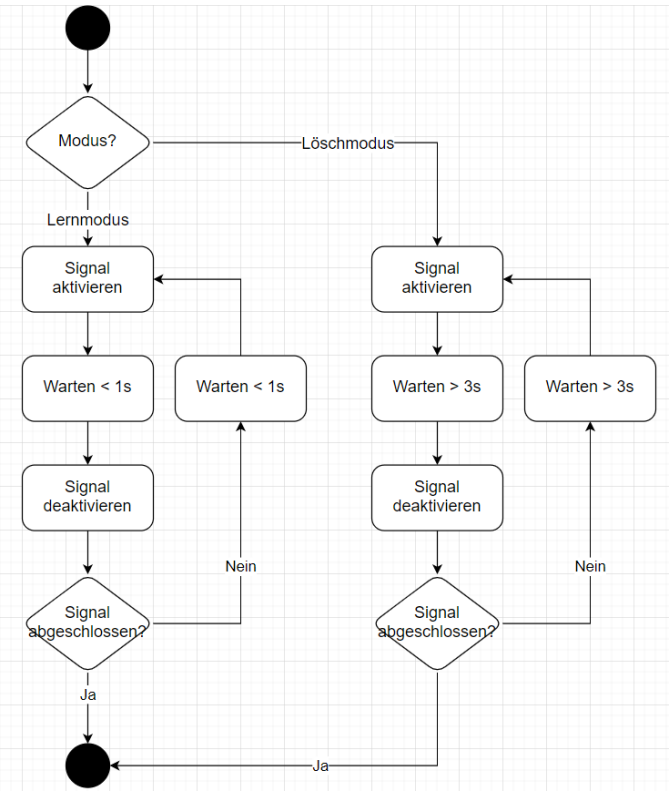

Abbildung 25: Ablaufdiagramm für Aktivierung der Lern- und Löschmodi des Funkmoduls

Die verschiedenen Modi werden dabei durch unterschiedliche Anzahl und Länge der Signalimpulse definiert [22, S. 2]. Für den Eintritt in einen der Lernmodi, wird das Signal mit einer Signaldauer von weniger als einer Sekunde übertragen. Die Anzahl der gesendeten Signale bestimmt dabei die Art des Lernmodus. Für Senden eines Löschsignals muss die Signallänge mindestens 3 Sekunden betragen [22, S. 2].

Durch diese langen Signalzeiten ist es hierbei besonders wichtig auf blockierende Wartefunktionen im Code zu verzichten. Auch hier wird deshalb wieder die Funktion so lange pausiert, bis der Timout der Funktion abgelaufen ist. Sie wird zwar in dieser Zeit zyklisch aufgerufen, es findet aber keine Verarbeitung statt, da sie direkt wieder verlassen wird.

```
270   int learn_RFControl(int mode){
285     if(mode >= 1 && mode <= 6){
287
288       // Wait for Timer
289       if(millis() < time_wait_Learn_RF && time_wait_Learn_RF > 0){
290         return 0;
291       }
```

Programmcode 16: Timeout Abfrage für Funktion learn_RFControl

Mit Aufruf der Funktion gibt es noch keine Timerbedinungen, weshalb bei erstem Eintritt in die Funktion keine Wartezeit abgewartet werden muss. Stattdessen wird das Signal aktiviert, indem der zugehörige Ausgang an der Porterweiterung auf Low gesetzt wird, da es sich hier um ein lowaktives Signal handelt.

Hier wird auch, abhängig vom übergebenen Modus die Wartezeit festgelegt, für welche das Signal aktiv sein soll. Damit der Ausgang nicht bei jedem zyklischen Aufruf neu beschrieben wird, wird das Setzen des Signals über die Variable Signal_active verriegelt. Dieses wird nur im Falle eines Fehlers in der I^2C-Kommunikation oder beim zurücksetzen des Signals wieder freigegeben. Zusätzlich wird im Falle eines Fehlers der Timer resettet, der Signalcounter ctr wieder zurückgesetzt und die Funktion mit einer Fehlerrückgabe

verlassen. Bei Erfolg beendet sich die Funktion mit Returnwert 0, welcher als pausierte Funktion interpretiert wird.

```
270   int learn_RFControl(int mode){
285     if(mode >= 1 && mode <= 6){
292
293       // Activate Signal
294       if(signal_active = false){
295         // Write to PIN 1 on GPIO B Expansion
296         signal_active = true;
297
298         // Set Timer
299         if(mode <= 4){
300           time_wait_Learn_RF = millis() + 100;
301         }
302         else{
303           time_wait_Learn_RF = millis() + 3100;
304         }
305
306         // lowactive Signal activate
307         rv = GPIO_Exp_WriteBit(GPIO_EXP_GPIOB, 1, LOW);
308         if(rv == ERROR){
309           signal_active = false;
310           time_wait_Learn_RF = 0;
311           ctr = 0;
312           return ERROR;
313         }
314         return 0;
315       }
```

Programmcode 17: Festlegung des Timeout und Aktivierung der Signalübermittlung für den Lern- / Löschmodus der Funkmoduls

Ist die Wartezeit abgelaufen, soll das Signal wieder zurückgesetzt werden. Das soll nur geschehen, wenn das Signal vorher aktiv war, um unnötige Schreibprozesse auf die Porterweiterung zu vermeiden. Hier wird als erstes das Signal wieder auf inaktiv gesetzt, um bei erneutem Aufruf der Funktion wieder am Anfang der Funktion starten zu können. Auch wird hier der Signalcounter erhöht, da ein weiterer Signalimpuls erfolgreich gesendet wurde. Dieses erfolgreiche Senden wird durch Rücksetzen des Pins auf High erreicht. Im Falle eines Fehlers bei der Kommunikation wird die Funktion mit Rückgabe eines Fehlers verlassen und alle Variablen auf den Initialisierungswert zurückgesetzt.

Als letztes wird gecheckt, ob das Senden des erwarteten Signals abgeschlossen ist. Die Anzahl der Impulse entspricht bei den Lernmodi dem Wert des Modus, die Löschmodi benötigen jedoch nur einen oder zwei Signalimpulse und werden vorher normiert. Wenn die Anzahl der bisher gesendeten Signale noch unter der erwarteten Anzahl gesendeter Signale liegt, wird der Timer neu konfiguriert und die Funktion mit Rückgabewert 0

verlassen. Dadurch wird die Funktion im nächsten Durchlauf mit aktivierter Wartezeit von vorne begonnen.

```
270   int learn_RFControl(int mode){
285     if(mode >= 1 && mode <= 6){
316
317       // Deactivate Signal
318       if(signal_active == true){
319         signal_active = false;
320         ctr++;
321
322         // lowactive Signal reset
323         rv = GPIO_Exp_WriteBit(GPIO_EXP_GPIOB, 1, HIGH);
324         if(rv == ERROR){
325           time_wait_Learn_RF = 0;
326           ctr = 0;
327           return ERROR;
328         }
329
330         int cmp_mode = mode;
331         if(cmp_mode >= 5){
332           cmp_mode -= 4;
333         }
334         if(ctr < cmp_mode){
335           // Signal not complete, continue
336           // Set Timer for Wait
337           if(mode <= 4){ time_wait_Learn_RF = millis() + 100; }
338           else{ time_wait_Learn_RF = millis() + 3100; }
339           return 0;
340         }
341         else{
342           // Signal complete, End Function
343           time_wait_Learn_RF = 0;
344           ctr = 0;
345           return SUCCESS;
346         }
347       }
```

Programmcode 18: Deaktivierung des Signals und Aktivierung eines neuen Signalimpulses für den Lern-/Löschmodus

Ist das Signal vollständig, wird die Funktion erfolgreich verlassen. Den Aufruf der Funktion übernimmt dabei der loop. Dieser überprüft, ob das Flag ISR_Learn_RF_Flag gesetzt ist. Solange das der Fall ist, wird die Funktion zyklisch aufgerufen. Sobald das Signal vollständig gesendet wurde, wird die Funktion check_RF_Acknowledge() aufgerufen. Diese soll anhand des LED-Signals, welches vom Funkmodul ausgegeben und vom Mikrocontroller eingelesen wird, erkennen, ob der

Modus erfolgreich aktiviert wurde. Ist dies der Fall, wird das Flag ersetzt und der Aufruf der Funktionen zur Aktivierung der Lernmodi beendet.

```
919  int check_RF_Acknowledge(int mode){
929    /*
930    Acknowledge Data Overview
931    Selection learn mode I: Light interrupts 1x every 2s
932    Selection learn mode II: Light interrupts 2x every 2s
933    Selection learn mode III: Light interrupts 3x every 2s
934    Selection learn mode IV: Light interrupts 4x every 2s
935    Selection erase mode I: Flashes permanently
936    */
937    static unsigned long time_wait_ACK = 0;
938
939    if(Learn_RF_ACK_Waiting == false){
940      // Start waiting for ACK, only operate once on first call
941      ISR_Learn_LED_CTR = 0;
942      time_wait_ACK = millis() + 2000;  // set timestamp for waiting tim
943      Learn_RF_ACK_Waiting = true;
944    }
945    unsigned long now = millis();
946    if(now >= time_wait_ACK){
947      // Timer is over, count captured Signals
948      if(mode == ISR_Learn_LED_CTR || ISR_Learn_LED_CTR >= 5){
949        // Captured valid ACK
950        time_wait_ACK = false;
951        ISR_Learn_RF_Flag = false;
952        Learn_RF_ACK_Waiting = false;
953        return SUCCESS;
954      }
955      else{
956        // wait for valid ACK
957        time_wait_ACK = false;
958        return 0;
959      }
960    }
961    else{
962      // Waiting for ACK
963      return 0;
```

Programmcode 19: Empfang und Auswerten des Acknowledge Signals nach Aktivierung eines Lern-/Löschmodus

Die Funktion zur Auswertung der Rückgabewerte erfolgt dabei durch Ermittlung der Frequenz der gesendeten Signale. Die Acknowledge Signale für die einzelnen Modi unterschieden sich nur in der Anzahl ihrer gesendeten Signalimpulse. Dabei bezieht sich die Anzahl der gesendeten Impulse immer auf 2 Sekunden [22, S. 2]. Die Funktion setzt also zu Beginn der Funktion den Counter auf 0 zurück und startet einen 2 Sekunden-Timer. Nach Ablauf dieses Timers wird wieder der Zähler ausgelesen und die gezählten Impulse

mit den erwarteten verglichen. Entspricht die Anzahl der gezählten Impulse denen des Modus, kehrt die Funktion erfolgreich zurück. Um zu verhindern, dass durch Timing Probleme oder ungünstige Signalflanken das erste Signal verfälscht ist, wird die Funktion zyklisch aufgerufen. Die Deaktivierung der Funktion nach einer bestimmten Anzahl versuche übernimmt der loop. Das Zählen der Signale und erhöhen des Counters übernimmt dabei die Interrupt Service Routine des GPIO Port Expanders. Dieser reagiert mit einem Interrupt auf jeden Lowpegel am entsprechenden Pin und erhöht den Counter in seiner ISR. So ähnlich läuft auch die Verarbeitung von aufgetretenen Fehlern ab, welche von dem Funkempfänger an den ESP32 gemeldet werden.

```
869   int check_RF_Error(){
876     /* Error Data Overview
882
883     if(RF_Error_Active == false){
884       RF_Error_Active = true;
885       time_wait_Error = millis() + 500;
886     }
887     unsigned long now = millis();
888     if(now >= time_wait_Error){
889       // Read Error Code
890       if(Learn_RF_Active_Flag == true){
891         if(ISR_RF_Error_CTR == 2 && ISR_Learn_RF_Mode >= 5){
892           // Entry could not be Errased from list
893           // Write to Website
894           RF_Connect_Return = "Fehlgeschalgen - Konnte nicht gelöscht we
895         }
896         if(ISR_RF_Error_CTR == 2 && ISR_Learn_RF_Mode <= 4){
897           // Entry already found in list
898           // Write to Website
899           RF_Connect_Return = "Fehlgeschalgen - bereits verbunden";
900         }
901         if(ISR_RF_Error_CTR == 3 && ISR_Learn_RF_Mode <= 4){
902           // Entry could not be added - maximum reached
903           // Write to Website
904           RF_Connect_Return = "Fehlgeschalgen - maximum erreicht";
905         }
906       }
907
908       RF_Error_Active = false;
909       ISR_RF_Error_CTR = 0;
910       return SUCCESS;
911     }
912     else{
913       // waiting for Error
914       return 0;
915     }
```

Programmcode 20: Einlesen und auswerten auftretender Fehler im Betrieb des Funkmoduls

In der Auswertungsfunktion der Interrupts der Port Erweiterung wird ein Counter erhöht, für jedes Fehlersignal, dass vom Funkempfänger gesendet wird. Auch hier wird das Signal über die Anzahl der gesendeten Impulse definiert [22, S. 2]. Wird ein Fehlersignal registriert, wird im Loop die Funktion check_RF_Error() ausgeführt. In dieser Funktion wird als erstes eine Wartefunktion aufgerufen. Diese hat die Aufgabe nach dem ersten Impuls, welcher zum Aufruf der Funktion geführt hat, auf weitere Signalimpulse zu warten. Danach werden die erhaltenen Fehlerinformationen ausgewertet. Zwei Impulse haben dabei mehrere Bedeutungen, je nach aktuell aktiviertem Lernmodus. Im Zusammenhang mit einem aktiven Löschmodus, bedeutet diese Sequenz, dass der gewünschte Eintrag nicht gelöscht werden konnte. In Zusammenhang mit einem der Lernmodi weist dieser Fehler darauf hin, dass die Fernbedienung bereits verbunden wurde [22, S. 2]. Drei Impulse hingegen bedeuten, dass der Adressspeicher des Moduls bereits gefüllt ist und keine weiteren Fernbedienungen mehr hinzugefügt werden können. Tritt einer dieser Fehler auf, soll das auf der Webseite für den Nutzer dargestellt werden. Zum Abschluss der Funktion wird der Fehlerzähler resettet, um für die nächsten Auftretenden Fehler keine verfälschten Daten zu erhalten.

Als nächstes soll die Software betrachtet werden, welche für die Kommunikation über den CAN1 Bus benötigt wird. CAN wurde bereits für die Übermittlung der Funksignale an das Steuergerät und den Erhalt von Befehlen zum Schalten der LED kurz genannt. Diese Funktionalitäten sollen nun im Folgenden betrachtet und erläutert werden, dabei soll es nicht nur um senden und empfangen von Nachrichten gehen, sondern auch die Verarbeitung der erhaltenen Daten betrachtet werden.

```
529
530  void init_can1(){
531
532    Serial.println("Init CAN");
533
534    CAN_cfg.rx_pin_id = gpio_num_t(CAN1_RX_PIN);
535    CAN_cfg.tx_pin_id = gpio_num_t(CAN1_TX_PIN);
536    CAN_cfg.speed = CAN_SPEED_500KBPS;
537    CAN_cfg.rx_queue = xQueueCreate(40, sizeof(CAN_frame_t));
538    if(ESP32Can.CANInit() == ESP_OK){
539      CAN1_active = true;
540      Serial.println("CAN1 started");
541    }
542    else{
543      CAN1_active = false;
544      Serial.println("Error starting CAN1");
545    }
546
547    define_CAN1_Messages();
548  }
```

Programmcode 21: Initialisierung des internen CAN-Controllers für CAN1

Die dafür notwendige Initialisierung findet auch für CAN wieder in der Funktion Setup() statt, welche immer einmal nach dem Bootvorgang abgearbeitet wird. Die Initialisierung für CAN ist dabei zweiteilig. Als erstes findet die Definition der Pins und des CAN-Objekts statt. Für die CAN-Kommunikation werden die Librarys <ESP32CAN.h> und <CAN_config.h> genutzt. Bei dieser wird ein Objekt vom Typ CAN_device_t mit Namen CAN_cfg erzeugt. Dieses speichert alle zur Kommunikation nötigen Einstellungen wie Pins und Busrate und ermöglicht den Zugriff auf empfangene und zu Sendende Nachrichten. So werden die beiden Pins für die Kommunikation auf die bereits vorher definierten Pins des ESP32 festgelegt. Über das Attribut speed wird die Busrate festgelegt, als mit welcher Geschwindigkeit der Bus kommuniziert. Diese beträgt beim vorliegenden System 500 Mbit/s. Um Nachrichten empfangen zu können muss ein Buffer für Nachrichten definiert werden. Das passiert über das Attribut rx_queue und die Funktion xQueueCreate(). Diese alloziert Speicherplatz, in welchem ankommende Nachrichten gespeichert werden können und erwartet dabei als Attribute die Anzahl an Speichereinheiten und die Größe der zu reservierenden Speichereinheiten. Die Größe der Speichereinheiten wird dabei als die Größe eines CAN-Nachrichten Objekts festgelegt, wovon maximal 40 im Buffer gespeichert werden sollen. Danach wird die bereits von der Library zur Verfügung gestellte Funktion CANInit aufgerufen, welche die Kommunikation für den internen TWAI-Controller aufbaut und ihn entsprechend konfiguriert. Ist diese Funktion erfolgreich, wird dies in Form der Variable CAN1_active gespeichert. Diese dient dazu, um keine Fehler durch Versuche auf einen inaktiven CAN-Bus zu senden zu erzeugen. Der zweite Teil der Initialisierungsfunktion betrifft nicht die Konfiguration für TWAI-Controller oder ESP32, sondern definiert die CAN-Nachrichten und Signale gemäß dem vorhandenen DBC-File. Da es sich nur um wenige Nachrichten und Signale handelt, werden diese händisch im C-Code definiert. Dafür wurden in einer eigenen Headerdatei Strukturen für Nachrichten und Signale definiert, welche die Eigenschaften der Signale und Nachrichten im C-Code abbilden.

```
18
19  struct CAN_Signal{
20    uint32_t length;
21    uint8_t polarity;
22    uint16_t factor;
23    uint16_t offset;
24    uint32_t start_bit;
25  };
26
27  struct CAN_Message{
28    uint32_t id;
29    uint8_t n_signals;
30    CAN_Signal* signals[8];
31  };
```

Programmcode 22: Definition der Strukturen für CAN-Nachricht und CAN-Signal

Zu den zu speichernden Daten für die Signale gehört dabei die Länge in Bits, die Polarität des gespeicherten Wertes, also ob dieser Wert als negativer Wert interpretiert werden kann. Zusätzlich wird ein Faktor gespeichert, mit welchem die ausgelesenen Daten multipliziert werden sollen, um zum Beispiel Kommazahlen übertragen zu können. Der Offset Wert wird verwendet, um den Wertebereich des Signals zu verschieben, da dieser auf den erhaltenen Wert aufaddiert wird. Zuletzt wird für jedes Signal noch sein Startbit innerhalb der Nachricht gespeichert. Die Nachricht hingegen speichert nur die Signale, welche der Nachricht zugeordnet wurden, sowie deren Anzahl. Die Nachrichten werden dabei als Zeiger auf ein Objekt gespeichert, um keine unnötigen Kopien der Objekte zu erzeugen. Die wichtigste Eigenschaft, welche eine Nachricht besitzt, ist ihre ID. Über diese wird jede Nachricht identifiziert. Innerhalb der Funktion define_CAN1_Messages() werden nun alle benötigten Signale und Nachrichten erstellt und die Signale den Nachrichten zugewiesen.

```
1452    Overall_Voltage.length = 8;
1453    Overall_Voltage.factor = 0.1;
1454    Overall_Voltage.offset = 40;
1455    Overall_Voltage.polarity = UNSIGNED;
1456    Overall_Voltage.start_bit = 48;
1457
1458    Battery_Voltage.id = 0x20;
1459    Battery_Voltage.n_signals = 4;
1460    Battery_Voltage.signals[0] = &Avg_Cell_Voltage;
1461    Battery_Voltage.signals[1] = &Max_Cell_Voltage;
1462    Battery_Voltage.signals[2] = &Min_Cell_Voltage;
1463    Battery_Voltage.signals[3] = &Overall_Voltage;
```

Programmcode 23: Definition des Overall_Voltage Signals und der Battery_Voltage Nachricht

Beispielhaft kann man das an dem Signal Overall_Voltage und der Nachricht Battery_Voltage erkennen. Mit den Übergebenen Parametern ergibt sich für das Signal ein Wertebereich von 0 bis 255 V, welcher mit 8 Bits darstellbar ist. Durch den Faktor von 0.1 wird der Wertebereich zugunsten der Genauigkeit auf 0 – 25.5V verkleinert. Um den geforderten Wertebereich, in welchem sich die Akkuspannung bewegen kann, abzubilden, wird nun noch ein Offset von 40V hinzugefügt, damit können Werte zwischen 40V und 65.V abgebildet werden. Dieses Signal wird anschließend der Nachricht Battery_Voltage hinzugefügt. Diese besitzt die ID 32 oder 0x20. Neben der Gesamtspannung des Akkus überträgt diese Nachricht noch drei weitere Signale, welche die Minimale, Maximale und Durchschnittliche Zellspannung abbilden. Diese Informationen können genutzt werden, um CAN-Frames zu bauen, welche an das Steuergerät gesendet werden sollen.

```
1332  int send_SOC_Request(){
1375    // Send CAN Message
1376    if(ESP_storage.getInt("RF_CAN_en", FALSE) == TRUE){
1377      //Send CAN-Message with SOC Request Command to VCU
1378      uint8_t data[0] = {};
1379      data[0] = 1 << SOC_Request.start_bit; // Set SOC Bit to HIGH
1380      int rv = send_CAN1_Message(VCU_Commands.id, data, 1); // Send CAN
1381
1382      if(rv == ERROR){
1383        Serial.println("Send CAN Message failed");
1384        return ERROR;
1385      }
1386    }
1387
1388    return SUCCESS;
1389  }
```

Programmcode 24: Senden des SOC-Request als CAN-Signal an die VCU

Das wird hier am Beispiel des SOC-Request Signals gezeigt, welches vom Mikrocontroller an das Steuergerät gesendet wird. Zum Senden wird ein Array festgelegt, wobei jeder Eintrag ein Datenbyte enthält. Die Position der Datenbits innerhalb des Bytes wird über das Startbit definiert, welches im entsprechenden Signalobjekt gespeichert ist. Dadurch, dass hier nur ein einzelnes Bit gesendet wird, können die Faktoren Offset und Faktor vernachlässigt werden. Das tatsächliche Senden der Nachricht passiert in der Funktion send_CAN1_Message(). Diese erwartet neben dem gerade erstellten Datenarray die ID der zu sendenden Nachricht und die Anzahl der zu Sendenden Bytes.

Innerhalb der Funktion wird zuerst die Korrektheit der Eingaben überprüft. Sollte hierbei ein Fehler auftreten, wird die Funktion mit einem Fehler verlassen. Darauf folgt der Aufbau des CAN-Frame, welcher durch die Library definiert wird und an die VCU gesendet werden soll. Hier werden die an die Funktion übergebenen Parameter an das CAN-Objekt übergeben. Dafür wird zuerst das Format der Nachricht definiert. In diesem Fall handelt es sich um eine Nachricht im Standardformat, ohne Extended ID oder andere Sonderformate. Als nächstes werden Identifier und Datenlänge übergeben, bevor die Daten aus dem übergebenen Array in den Frame übertragen werden. Über die Funktion CANWriteFrame() wird nun die Nachricht gebaut, in den Sendebuffer geschrieben und schließlich vom CAN-transceiver auf den Bus gesendet. Ist diese Funktion erfolgreich,

wird die Funktion erfolgreich verlassen, in anderen Fällen unter Rückgabe eines Fehlers beendet.

```
1391  int send_CAN1_Message(int id, uint8_t* data, int len){
1392  /**
      */
1400    if(len > 8){
1401      Serial.println("Data Length too long");
1402      return ERROR;
1403    }
1404
1405    CAN_frame_t tx_frame;
1406    tx_frame.FIR.B.FF = CAN_frame_std;
1407    tx_frame.MsgID = id;
1408    tx_frame.FIR.B.DLC = len;
1409
1410    for(int i=0; i<len; i++){
1411      tx_frame.data.u8[i] = data[i];
1412    }
1413
1414    // Send Message
1415    if(CAN1_active == true){
1416      if(ESP32Can.CANWriteFrame(&tx_frame) == ESP_OK){
1417        return SUCCESS;
1418      }
1419      else{
1420        Serial.println("Error sending CAN1 Message");
1421        return ERROR;
1422      }
1423    }
1424    else{
1425      Serial.println("CAN1 not active");
1426      return 0;
1427    }
1428
1429  }
```

Programmcode 25: Funktion zum Senden einer CAN-Nachricht auf CAN1

Der Empfang und die Auswertung empfangener Nachrichten sind dagegen in mehrere Abschnitte aufgeteilt. Den Empfang der Nachrichten übernimmt dabei der Controller selbstständig und schreibt alle empfangenen Nachrichten in einen Empfangsbuffer, welcher bei der Initialisierung des CAN-Objekts definiert wurde. Um Nachrichten verarbeiten zu können, muss nun lediglich der Speicher zyklisch ausgelesen werden. Das passiert innerhalb der Loopfunktion. Der Mikrocontroller reagiert dabei nicht auf einen Interrupt oder ähnliches, sondern versucht bei jedem Durchlauf der Loop-Funktion den Speicher auszulesen. Dafür wird die Funktion process_CAN1() aufgerufen. Diese liest mit der Funktion xQueueReceive den Buffer des CAN-Controllers aus. Dabei muss ihr dieser Speicherbereich übergeben werden zusammen mit einem CAN-Frame Objekt, in welches

der Inhalt der ausgelesenen Nachricht gespeichert werden soll. Zudem wird ein Timeout in Ticks hinzugefügt, welches angibt, wie lange die Funktion auf neue Nachrichten warten soll, bevor sie eine Nachricht ausliest. Wurde im Buffer eine Nachricht empfangen, gibt die Funktion pdTRUE zurück und die Verarbeitung des erhaltenen CAN-Frame kann beginnen.

```
651  int process_CAN1(){
652    // Process incoming CAN1 Messages
653    CAN_frame_t rx_frame;
654
655    // Check for IDs and process Data based on
656    // Versuchen, eine Nachricht aus der Queue zu holen
657    if (xQueueReceive(CAN_cfg.rx_queue, &rx_frame, 0) == pdTRUE) {
658
659      int identifier = rx_frame.MsgID;
660
661      if(Power_Data.id == identifier){
662        // Power Data Message
663        process_CAN_PowerData(&rx_frame);
664      }
665      else if(Battery_Temperature.id == identifier){
666        // Battery Temperature Message
667        process_CAN_BatteryTemperature(&rx_frame);
668      }
669      else if(Battery_Voltage.id == identifier){
670        // Battery Voltage Message
671        process_CAN_BatteryVoltage(&rx_frame);
672      }
673      else if(Option1_Commands.id == identifier){
674        // VCU Commands Message
675        process_CAN_Option1Commands(&rx_frame);
676      }
677      else{
678        // Unknown Message
679        Serial.println("Unknown CAN Message");
680        return 0;
681      }
682    }
683
684    return 0;
685  }
```

Programmcode 26: Verarbeitung der empfangenen Nachrichten über CAN1

Dafür wird lediglich die ID der Nachricht aus dem Objekt ausgelesen. Diese wird dann mit IDs der Nachrichten verglichen, welche verarbeitet werden sollen, um jede Nachricht entsprechend ihrer Anforderungen und enthaltenen Signale auswerten zu können. Stimmt die ausgelesene ID mit einer der erwarteten überein, wird für diese Auswertungsfunktion aufgerufen. Diese Auswertung soll beispielhaft anhand einer

erhaltenen Battery_Voltage Nachricht gezeigt werden. In diesem Fall würde die Funktion process_CAN_BatteryVoltage() aufgerufen werden. Diese erhält für alle benötigten Signale aus einer Nachricht den Wert anhand der für jedes Signal spezifizierten Einstellungen über die Funktion decodeSignal() und verarbeitet diesen Wert entsprechend. Im Falle der Nachricht Battery_Voltage ist nur das Signal Overall_Voltage interessant. Dessen Wert wird auf dem Display und auf der Livedatenseite dargestellt. Dafür wird der Wert in eine globale Variable geschrieben, welche nach einer Aktualisierung von Webseite und Display dargestellt wird. Die Funktion decodeSignal erwartet für die Auswertung neben dem erhaltenen CAN-Frame auch das Signalobjekt, welches aus der Nachricht ausgelesen werden soll. Diese Funktion erzeugt zuerst aus den Datenbytes, welche in einem Array gespeichert sind, einen großen Wert. Das ist sehr hilfreich für Signale, welche länger als ein Byte sind oder Daten, die über Bytegrenzen hinweg gespeichert sind. Um das zu erreichen, werden die Bytes gemäß ihrer Position immer um Vielfache von 8 bitweise geschoben und mit den Bytes vorher verbunden.

```
1622
1623  int decodeSignal(CAN_Signal signal, CAN_frame_t* frame){
1624    int data = 0;
1625
1626    // Combine all data arrays to one value
1627    for(int i = 0; i < frame->FIR.B.DLC; i++){
1628      data |= frame->data.u8[i] << (i * 8);
1629    }
1630
1631    // Read Speed from Data
1632    // generate mask
1633    int mask = 0;
1634    for(int i=0; i<signal.length; i++){
1635      mask = (mask << 1) | 0x01;
1636    }
1637
1638    // Process all signals
1639    int result = data >> signal.start_bit & mask;
1640
1641    // Apply factor and offset
1642    float value = result * signal.factor + signal.offset;
1643
1644    return value;
1645  }
```

Programmcode 27: Ermittlung der übertragenen Daten aus einem CAN-Objekt

Um nur die Bits zu betrachten, welche auch zum passenden Signal gehören, wird als erstes eine Maske generiert. Diese Maske besitzt nur in dem Bereich Einsen, in denen das Signal gespeichert ist. Diese Maske wird über die gespeicherte Signallänge definiert. Damit wird das Signal maskiert, sodass alles Bits gelöscht werden, die nicht zu den Daten des Signals gehören und die Daten um den Wert des Startbits zurückgeschoben, wodurch

der Wert der Daten nun dem Wert des Signals entspricht. Um den korrekten Wert des Signals zu erhalten, wird es noch mit dem definierten Faktor multipliziert und anschließend der Offset-Faktor aufaddiert. Dieser Wert wird zurückgegeben. Dieser Ablauf ist für alle erhaltenen Nachrichten derselbe bis auf die Nachricht Option1_Commands. Alle anderen Nachrichten enthalten nur Signale, deren Daten visuell auf dem Display oder der Webseite dargestellt werden sollen. Die Nachricht Option1_Commands enthält hingegen Befehle vom Steuergerät an die Erweiterungsplatine, welche entsprechend verarbeitet werden müssen. Dabei handelt es sich um die Befehle zur Steuerung der Status-LED. Dabei wird zwischen zwei Signalen unterschieden. Das erste der beiden Signale übermittelt einen Wert, welcher Angibt, wie oft die LED blinken soll. Das wird genutzt, wenn die VCU mit einem exakten Wert auf die SOC-Anfrage antwortet. Dieser Wert wird ebenfalls in eine Globale Variable geschrieben, welche als Zähler fungiert. Das zweite Signal ist Enable-Bit, um die LED so lange blinken zu lassen, bis die VCU das Blinken wieder beendet. Die Bearbeitungsfunktion setzt hierfür ein Flag, welches das Blinken für andere Funktionen indiziert.

Zusätzlich aktivieren beide Signale einen Timer. Dieser Timer dient als Taktgeber für das Blinken der LED. Dieser Timer wird bereits zu Beginn des Programmablaufs im Setup initialisiert und gestartet, jedoch noch nicht aktiviert. Der Timer mit seinen Einstellungen wird über die Funktion timerBegin() gestartet und initialisiert. Der Takt des Timer ergibt sich über den Prescaler und den Wert im Auto Reload Register (ARR).

```
450  void init_Timer(){
451
452    /*************** Timer 1 - LED Flashing via CAN ********************
453    // Timer 1, Prescaler 8000 -> 1 tick = 0.01 ms, countUp
454    TIM_LED_Flashing = timerBegin(1, 8000, true);
455
456    // init Interrupt for Timer overflow
457    timerAttachInterrupt(TIM_LED_Flashing, TIM_LED_Flashing_overflow, tr
458
459    // set Timer-Alarm: 500 ms = 0.5 Sekunden -> Period = 1 sec
460    timerAlarmWrite(TIM_LED_Flashing, 5000, true); // true = auto-reload
461    timerAlarmDisable(TIM_LED_Flashing);     // will be activated by ISR
462    /*******************************************************************
463  }
```

Programmcode 28: Festlegung und Berechnung der Timerlaufzeit

Über den Prescaler wird die Timerauflösung festgelegt und darüber die Zählgeschwindigkeit. Dabei handelt es sich lediglich um einen Vorteiler, welcher die Eingangsfrequenz verkleinert. Das ARR legt den Wert fest, bei dem der Zähler resettet wird und eine Funktion ausführt. Das bestimmt die Laufzeit des Timers. Die beiden Werte lassen sich dabei über folgende Formel berechnen:

$$F_{Timer} = \frac{Clock_{In}}{Prescaler * ARR}$$
$$ARR = \frac{Clock_{In}}{F_{Timer} * Prescaler}$$

Für die Frequenz, welche der Timer ausgeben soll, wird 2 Hz festgelegt. Für den Prescaler bietet es sich an Vielfache des Clock-Taktes zu wählen, um gerade Werte für die Eingangsfrequenz zu erhalten.

$$ARR = \frac{80MHz}{2Hz * 8000} = 5000$$

Der Wert des Prescalers wird direkt beim Start des Timers festgelegt. Der Wert des ARR wird über die Funktion timerAlarmWrite() festgelegt. Hier wird neben dem Timerobjekt der Wert des ARR übergeben, bei dem der Timer einen Alarm auslösen soll. Dieser Alarm löst einen Interrupt aus. Dieser wird über die Funktion timerAttachInterrupt() konfiguriert. Hier wird festgelegt, welche Funktion als ISR bei Erreichen des ARR ausgeführt wird. Das ist die Funktion TIM_LED_Flashing_overflow(). Die Funktion übernimmt dabei sowohl das dauerhafte Blinken als auch die Kontrolle über eine Bestimmte Anzahl an Impulsen für die LED. Zuerst wird wieder die Priorisierung der LED-Signalquellen beachtet. So soll die LED nur blinken, wenn das Signal gerade nicht durch die Fernbedienung überschrieben wird.

```
1645  void IRAM_ATTR TIM_LED_Flashing_overflow(){
1647
1648    if(ID_cur_state = OFF){
1649      // Constant Flashing
1650      if(LED_Flashing_Active == true){
1651        // Flash LED permanently
1652        if(led_state == OFF){
1653          // Set LED on
1654          ISR_LED_Signal_Flag = ON;
1655          led_state = ON;
1656        }
1657        else{
1658          // Set LED off
1659          ISR_LED_Signal_Flag = OFF;
1660          led_state = OFF;
1661        }
1662      }
```

```
1663        else{
1664          if(LED_Flashing_CTR > 0){
1665          // Flash LED
1666            if(led_state == OFF){
1667              // Set LED on
1668              ISR_LED_Signal_Flag = ON;
1669              led_state = ON;
1670              LED_Flashing_CTR -= 1;
1671            }
1672            else{
1673              // Set LED off
1674              ISR_LED_Signal_Flag = OFF;
1675              led_state = OFF;
1676            }
1677          }
1678          else{
1679            // Stop Flashing
1680            ISR_LED_Signal_Flag = OFF;
1681            timerAlarmDisable(TIM_LED_Flashing);       // stop Timer
1682          }
```

Programmcode 29: Timerfunktion zum Dauerhaften und gesteuertem Blinken der LED

Ist das Flag für das dauerhafte Blinken gesetzt wird der Counter zur Bestimmung der Anzahl der Impulse missachtet und der Zustand der LED wird bei jedem Erreichen des Timer-ARR gewechselt. Ist das Flag nicht gesetzt, prüft die Funktion, ob der Counter noch Impulse der LED erwartet. Ist das der Fall wird auch hier mit jedem Aufruf der Funktion der Zustand der LED gewechselt. Zusätzlich wird jedoch der CTR bei jedem Leuchten der LED dekrementiert. Ist die korrekte Anzahl an Impulsen abgearbeitet, wird der Timeralarm deaktiviert, sodass der Timer keinen Interrupt mehr auslöst und die Funktion nicht mehr aufgerufen wird, bis wieder ein Signal via CAN erhalten wird. Da es sich hier um einen Funktionsaufruf innerhalb einer Interrupt Service Routine handelt wird nicht direkt die Funktion Status_LED_ON() oder Status_LED_OFF() aufgerufen, welche den Transistor des LED-Treibers schaltet, da diese die I^2C Kommunikation zur GPIO-Porterweiterung übernimmt. Diese ist innerhalb einer ISR nicht zulässig, da sie für die Dauer der Kommunikation alle anderen Funktionen blockieren würde. Stattdessen wird hier das ISR_LED_Signal_Flag gesetzt, welches regelmäßig im Loop abgefragt wird.

Neben CAN-Bus 1wird auch CAN-Bus 2 für Kommunikation genutzt. Da hierfür ein eigener externer CAN-Controller nötig wurde, unterscheiden sich die Abläufe zum Empfangen und Senden von Nachrichten etwas und auch die Konfiguration des Controllers fällt umfangreicher aus als bei CAN1. Auch hier findet die Initialisierung im Setup() statt. Die Funktion init_can2() übernimmt dabei die Konfiguration des CAN-Controllers. Die Kommunikation mit dem Controller findet dabei über SPI statt. Um die Initialisierung des Controllers vornehmen zu können, muss er zuerst in den Config Modus versetzt werden.

Das passiert, indem in das CAN_Control-Register (CANCTRL) der Wert 0x80 geschrieben wird [24, S. 59].

```
685   void init_CAN2(){
692
693     // Initialize CAN2 Module via SPI
694     // Set CAN2 Module to Config Mode
695     write_SPI_Register(CAN2, CAN2_CANCTRL,0x80); // Set Config Mode
696     delay(200);
697
698     // Check for Controller in Config Mode
699     // Read CANSTAT Register [7:5] -> Mode
700     mode = read_SPI_Register(CAN2, CAN2_CANSTAT) >> 5;
701     if(mode == 0x04){
702       Serial.println("CAN2 in Config Mode");
703       connected = true;
704     }
705     else{
706       // Error Handling
707       Serial.println("Error: CAN2 not in Config Mode");
708       connected = false;
709       CAN2_active = false;
710     }
```

Programmcode 30: Aktivierung des Konfigurationsmodus für den CAN2-Controller

Der Chip quittiert diesen Konfigurationsmodus, sobald er eingetreten ist, indem er im Register CAN_Status (CANSTAT) die letzten drei Bits auf den Wert 0x4 setzt [24, S. 61]. Die Konfigurierung wird nur fortgesetzt, wenn sich der Chip im korrekten Modus befindet. Das wird über die Variable connected für den weiteren Ablauf der Initialisierung gespeichert. Um sicherzustellen, dass der Chip genug Zeit hat sich in den Modus zu versetzen, wird in delay eingesetzt. Dieses stellt hier kein Problem dar, da diese Funktion nur einmal zu Beginn des Programmablaufs aufgerufen wird und daher die zyklischen Abläufe im loop nicht blockiert. Wurde der Chip erfolgreich in den korrekten Modus versetzt werden die entsprechenden Initialisierungen vorgenommen. Die Bitwerte, welche in den Registern Configuration 1 bis 3 (CNF1 ... 3) festgelegt werden, bestimmen das Timing des Busses und der Kommunikation auf Bitebene. Eng damit verbunden ist die Synchronisation der einzelnen Busknoten, da bei CAN kein Clocksignal übermittelt wird. Jedes Bit ist dabei in einzelne Segmente aufgeteilt, wie in Abbildung 26 dargestellt ist. Alle nachfolgenden Formeln und Aussagen stammen aus dem Datenblatt des Bauteils [24, S. 39 – 43].

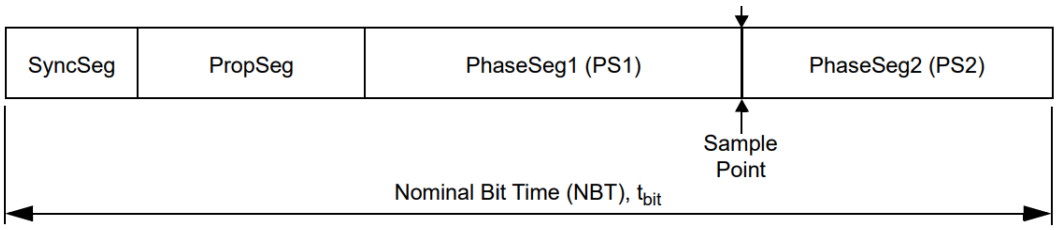

Abbildung 26: Unterteilung eines Bit in Segmente bei CAN-Übertragung, aus [24, S. 29]

Die Gesamtzeit ist die Nominal Bit Time t_{bit}. Diese wird durch die Busrate festgelegt. Für den vorliegenden CAN-Bus beträgt die Baudrate

$$f_{bit} = 500 \frac{kbit}{s}$$

Damit berechnet sich für t_{bit}:

$$t_{bit} = \frac{1}{f_{bit}} = \frac{1}{500000} = 2\mu s$$

t_{bit} setzt sich dabei zusammen aus den Einzelzeiten der Segmente, welche für eine fehlerfreie Kommunikation t_{bit} nicht überschreiten dürfen.

$$t_{bit} = t_{syncSeg} + t_{PrSeg} + t_{PS1} + t_{PS2}$$

Die Definition der Längen der einzelnen Segmente findet dabei in Vielfachen des Zeitabschnitts T_Q statt. Das ist die kleinste Zeiteinheit, mit welcher der Controller arbeiten kann. Der Wert von T_Q ergibt sich dabei aus der Frequenz des Oszillators $f_{Osz} = 8\ MHz$.

$$T_Q = 2 * (BRP + 1) * \frac{1}{F_{Osz}}$$

BRP ist dabei der Baud Rate Prescaler, welcher die Auflösung von TQ und Dauer von T_Q bestimmt. Für BRP = 0 ergibt sich für T_Q folgender Wert:

$$T_Q = 2 * (0 + 1) * \frac{1}{8\ MHz} = \frac{2}{8\ MHz} = 250\ ns$$

Für die Gesamtlänge ergibt sich somit

$$t_{bit} = x * T_Q$$
$$x = \frac{t_{bit}}{T_Q} = \frac{2\mu s}{250ns} = 8$$

Das erste Segment, welches innerhalb dieser 8 $T_Q$stattfinden muss, ist das Synchronization Segment. Um Phasenverschiebungen zwischen den Oszillator-frequenzen der einzelnen Knoten auf dem Bus auszugleichen, muss jeder CAN-Controller in der Lage sein, sich mit der relevanten Signalphase des eingehenden Signals zu synchronisieren. Dieses Segment ist auf die Länge von 1 T_Q festgelegt. Das Segment, dessen Länge als nächstes festgelegt wird, ist Phase Segment 2. Dieses ist vorhanden, um Edge Phase Errors auf dem Bus zu auszugleichen. Dieses Segment beginnt immer mit dem Sample Point. Das ist der Punkt an dem der Logikpegel gelesen und ausgewertet wird. Dieser sollte immer bei etwa 70% der Gesamtzeit liegen. Damit ergibt sich für t_{PS2}:

$$t_{PS2} = 8\ T_Q * (1 - 70\%) = 8\ T_Q * 30\% = 2.4\ T_Q$$

Da T_Q nur als ganzzahliges Vielfaches auftreten darf und t_{PS2} mindestens $2.4\ T_Q$ betragen muss, wird $t_{PS2} = 3\ T_Q$ festgelegt.

Für die Festlegungen der Längen von Propagation Segment und Phase Segment müssen zudem eine Regel erfüllt sein:

$$t_{PrSeg} + t_{PS1} \geq t_{PS2}$$

Mit $t_{PS2} = 3\ T_Q$ und $t_{sync} = 1\ T_Q$ bleiben für die restlichen beiden Segmente noch

$$t_{PrSeg} + t_{PS1} = t_{bit} - \left(t_{sync} + t_{PS2}\right)$$
$$t_{PrSeg} + t_{PS1} = 8\ T_Q - \left(1\ T_Q + 3\ T_Q\right) = 8\ T_Q - 4\ T_Q = 4\ T_Q$$

Damit ist die Bedingung $4\ T_Q \geq 3\ T_Q$ ebenfalls erfüllt.

Zuletzt müssen noch die Zeiten für Propagation und Phase Segment 1 festgelegt werden. Diese beiden müssen zusammen $4\ T_Q$ betragen, um die Gesamtzeit auszunutzen. Da die Laufzeit für Phase Segment 1 meist größer gewählt wird als die vom Propagationsegment, ergeben sich die Zeiten zu

$$t_{PrSeg} < t_{PS1}$$
$$t_{PS1} = 3\ T_Q$$
$$t_{PrSeg} = 1\ T_Q$$

Zusätzlich zu den Segmentlängen muss die Synchronization Jump Width (SJW) festgelegt werden. Diese passt den Bit-Takt nach Bedarf um $1 - 4\ T_Q$ an, um die Synchronisation mit der übertragenen Nachricht aufrechtzuerhalten. Zusätzlich gilt für SJW die Bedingung:

$$t_{PS2} > SJW$$

Typischerweise wird SJW bei stabiler Taktquelle auf $1\ T_Q$ festgelegt.

Die Initialisierung des Chips mit diesen Werten findet in den Registern Configuration 1 bis 3 (CNF 1… 3) statt. In CNF1 werden die Bits 6 und 7 zur Speicherung des Wertes von SJW verwendet. Dabei entspricht der Wert im Register SJW – 1. Für einen Wert von $1\ T_Q$ werden die Bits mit 00 beschrieben. Die Bits 0 bis 5 speichern den Prescaler BRP. Dieser wurde für die obere Rechnung als 0 festgelegt [24, S. 44]. Damit ergibt sich der Registerwert zu 0x00.
In CNF2 werden die Werte für Propagation Segment und Phase Segment festgelegt. Auch hierbei gilt, dass der im Register gespeichert Wert $t_{PS1} - 1 = 2$ und $t_{PrSeg} - 1 = 0$ entspricht. Die Länge des Propagation Segment wird dabei in die Bits 0 bis 2 eingetragen, die des Phase Segment 1 in die Bits 3 – 5. Bit 6 wird als Konfigurationsbit für den Sample Point genutzt. Wird dieses Bit auf Eins gesetzt, wird am Sample Point Drei mal abgetastet, liegt das Bit auf 0 nur einmal [24, S. 44]. Diese häufigere Abtastung hat den Vorteil das Fehler minimiert werden, da bis zu einem Fehler ignoriert werden kann. Bit 7 wird verwendet, um die Festlegung des Zeitwertes für Phase Segment 2 zu bestimmen. Wird

dieses Bit auf 0 gesetzt, wird $t_{PS2} > t_{PS1} > 2\ T_Q$ festgelegt. Mit dem Bit auf 1 wird der Wert verwendet, welcher für die Länge des Phase Segment 2 im CNF3 – Register festgelegt wird [24, S. 45]. Für die aktuelle Variante soll der Wert über das CNF3 Register festgelegt werden, sodass der Wert auf 1 gesetzt wird. So ergibt sich folgender Wert für das Register:

$$1 \mid 1 \mid 010 \mid 000 = 0xD0$$

```
685   void init_CAN2(){
713     if(connected == true){
714       // Set Bitrate to 500 kbps
715       // For 8MHz Clock
716       write_SPI_Register(CAN2, CAN2_CNF1, 0x00); // SJW=1, BRP=0 -> 500
717       write_SPI_Register(CAN2, CAN2_CNF2, 0xD0); // BTLMODE=1, SAM=1, PH
718       write_SPI_Register(CAN2, CAN2_CNF3, 0x02); // PHSEG2=3
719
720       // Enable Interrupts
721       write_SPI_Register(CAN2, CAN2_CANINTE, 0xA2); // Enable all interr
722
723       // Set Normal Mode
724       write_SPI_Register(CAN2, CAN2_CANCTRL, 0x00); // Set Normal Mode
725       delay(200);
```

Programmcode 31: Initialisieren des CAN-Controller mit den berechneten Werten für die Zeitsegmente

Wie bereits angesprochen muss nun noch der Wert für Phase Segment 2 in CNF3 gespeichert werden. Dafür werden die Bits 0 bis 2 beschrieben. Auch hier wird als Registerwert $t_{PS2} - 1$ eingetragen.

Zusätzlich zu den Konfigurationen für Timing und Synchronisation müssen auch die Einstellungen für Interrupt Erzeugung übergeben werden. Dabei wird über das Register CAN_Interrupt_Enable (CANINTE) verschieden Interruptquellen aktiviert. Für das vorliegenden Beispiel sollen Interrupts für Receive Buffer 0 Full, Receive Buffer 1 Full, Error Interrupt und Message Error Interrupt freigeschalten werden [24, S. 51]. Als letztes muss der Controller für normalen Betrieb wieder in den Normal Mode zurückgesetzt werden. Dafür wird wie bei der Aktivierung des Config Modus das CANCTRL-Register beschrieben, wobei der Wert 00 den korrekten Modus aktiviert.

Um diese Konfigurierungen der Register vornehmen zu können werden Funktionen benötigt, welche per SPI-Daten in die Register schreiben oder daraus lesen. Die dafür geschriebenen Funktionen sind write_SPI_Register() und read_SPI_Register(). Die Funktionen nutzen dabei die bereits vorhandenen Funktionen aus der SPI-Library zur Kommunikation auf dem Bus. So beginnt die Kommunikation immer mit dem Aktivieren des entsprechenden Chips über den Chip Select. Das übernimmt die Funktion SPI_select(), welche den übergebenen Chip über die passende CS-leitung aktiviert.

```
1687  void write_SPI_Register(int chip, uint8_t reg, uint8_t value){
1688    // Write single Register via SPI
1689    SPI_select(chip);
1690    my_SPI.transfer(0x03);      // Send Write Command
1691    my_SPI.transfer(reg);       // Send Register Address
1692    my_SPI.transfer(value);     // Send Value
1693    SPI_deselect();
1694    delay(10);
1695  }
1696
1697  uint8_t read_SPI_Register(int chip, uint8_t reg){
1698    // Read single Register via SPI
1699    SPI_select(chip);
1700    my_SPI.transfer(0x02);      // Send Read Command
1701    my_SPI.transfer(reg);       // Send Register Address
1702    uint8_t value = my_SPI.transfer(0x00); // Send Dummy Byte to receive
1703    SPI_deselect();
1704    delay(10);
1705    return value;
1706  }
```

Programmcode 32: SPI-Kommunikation zum Lesen und Schreiben von Registern im CAN2-Controller

Als erstes wird der Befehl für Lesen oder Schreiben übermittelt. Das passiert über die Funktion transfer() aus der SPI-Library. Auf den Befehl folgt die zu bearbeitende Registeradresse. Ab hier unterschieden sich beide Funktionen. Während in der Schreibfunktion nun der Registerwert gesendet wird, schickt die Lesefunktion ein Dummy Byte, woraufhin der Chip mit dem Inhalt des abgefragten Registers antwortet. Als letztes muss der Chip Select wieder zurückgenommen werden, sodass eine korrekte Kommunikation für den nächsten Chip sichergestellt werden kann.

Zusätzlich müssen im Rahmen der Initialisierung, wie bei CAN1, die benötigten Signale und Nachrichten definiert werden. Da CAN2 nur zur Kommunikation mit dem RFID-Reader verwendet wird, benötigt die Option 1 lediglich eine Nachricht mit zwei Signalen. Dabei handelt es sich um die auf der Karte gespeicherte Kundennummer und den Fahrmodus, welcher mit der Karte aktiviert werden soll. Der Ablauf zum Auslesen von Daten ist dabei unterschiedlich zu CAN1. Zum einen wird hier nicht jeden Durchlauf der Empfangsbuffer auf neue Nachrichten geprüft, sondern der CAN-Controller sendet ein Signal an den Mikrocontroller, welches dort einen Interrupt auslöst. In der dazugehörigen ISR wird dann ein CAN2-Flag gesetzt. Auf dieses reagiert der Loop mit dem Aufruf der Funktion process_CAN2(). Diese übernimmt die Auslesen und die Verarbeitung der erhaltenen Daten. Das Auslesen von Nachrichten funktioniert dabei genau wie die Konfigurierung über SPI. Als erstes wird geprüft, ob der aufgetretene Interrupt durch eine Nachricht im Receive Buffer ausgelöst wurde. Dafür wird das CAN_Interrupt_Flag-Register (CANINTF) ausgelesen. Der Interrupt für RX0 Buffer 0 wird durch Bit 0 indiziert,

weswegen der Registerwert entsprechend maskiert wird. Ist ein Interrupt aufgetreten, welcher auf eine neue Nachricht im Speicher hinweist, muss der Interrupt wieder zurückgesetzt werden, um bei einem erneuten Aufruf den Inhalt nicht zu verfälschen.

```
815  void process_CAN2_Message(){
816    // Check for Message in Buffer 0
817    // Check RX Buffer 0 Full Flag [bit:0]
818    if(read_SPI_Register(CAN2, CAN2_CANINTF) & 0x01){
819      // Clear Interrupt Flag
820      write_SPI_Register(CAN2, CAN2_CANINTF, 0x00);
821
822      // Message in Buffer 0
823      // Read Identifier from Controller
824      int identifier = (read_SPI_Register(CAN2, CAN2_RXB0SIDH) << 3);
825      identifier |= (read_SPI_Register(CAN2, CAN2_RXB0SIDL) >> 5);
826
```

Programmcode 33: Auslesen der Interrupts und Ermittlung der Message ID für CAN2

Anders als bei CAN1 gibt es hier leider kein CAN-Frame Objekt, aus welchem man ID und Daten auslesen kann, sondern diese Informationen müssen via SPI aus den Registern ausgelesen werden. Stattdessen wird die ID in den Registern Receive Buffer 0 Standard Identifier High (RXB0SIDH) und Receive Buffer 0 Standard Identifier Low (RXB0SIDL) gespeichert. RXB0SIDH speichert dabei die ersten Bits 10 - 3 des 11 Bit langen Identifiers, RXB0SIDL die Bits 2 – 0 [24, S. 63]. Um diese nun zu einem Wert zusammenzuführen, wird der Wert aus RXB0SIDH um 3 Bits verschoben und mit den 3 fehlenden Bits verbunden. Diese Bits sind im Register RXB0SIDL an den Positionen 7 – 5 gespeichert, weswegen der Registerwert zuerst auf die Ersten 3 Bits verschoben werden muss. Diese ausgewertete ID kann nun mit der erwarteten Nachricht verglichen werden. Ist die ID korrekt, wird mit der Auswertung und Verarbeitung der Daten begonnen. Dafür wird zuerst der DLC-Wert ausgelesen, welcher angibt, wie viele Bytes in der Nachricht übertragen wurden. Diesen Wert findet man im Receive Buffer 0 Data Length Code Register (RXB0DLC). Um den korrekten DLC zu erhalten, dürfen nur die ersten vier Bit betrachtet werden, sodass der Registerwert entsprechend maskiert wird. Mit diesem DLC-Wert können nun die Daten aus dem Speicher ausgelesen werden. Diese sind byteweise in den Registern Receive Buffer 0 Data 0 bis 7 (RXB0D0...7) gespeichert [24, S. 63]. Um aus diesen Bytes einen großen Wert zu erhalten, welcher anschließend ausgewertet werden kann, werden nacheinander die Datenbytes ausgelesen und höherwertige Bytes vor den bisher bereits ausgelesenen positioniert. Das wird durch Verschiebung in Bytes erreicht. Die Register bieten dabei den Vorteil, dass sie konsekutive Adressen besitzen, sodass die Startadresse inkrementiert werden kann, um die Folgebytes auszulesen.

```
815  void process_CAN2_Message(){
818    if(read_SPI_Register(CAN2, CAN2_CANINTF) & 0x01){
826
827      if(identifier == RFID_Data.id){
828        // RFID Message recieved
829        // Read Data Length Code
830        int dlc = read_SPI_Register(CAN2, CAN2_RXB0DLC) & 0x0F;
831
832        // Read Data Bytes
833        uint64_t data = 0;
834        for(int i=0; i<dlc; i++){
835          data = data | (read_SPI_Register(CAN2, CAN2_RXB0D0 + i);
836          data = data << (i * 8));
837        }
838        // Process Data
839        // generate mask for data
840        uint64_t mask = ~(0xFFFFFFFFFFFFFFFF << Customer_ID.length);
841
842        // Extract Signals
843        uint64_t Customer_identifier = (data >> Customer_ID.start_bit);
844        Customer_identifier &= mask;
845
846        // Handle data
847        if(Customer_identifier == MASTER_ID){
848          // Master ID detected
849          // Activate Master Mode
850          Master_Mode_active = true;
851        }
852        else{
853          Master_Mode_active = false;
854        }
855      }
```

Programmcode 34: Ermittlung der Daten aus den Registern des CAN2-Controllers

Um die einzelnen Signale aus den Daten auszulesen, wird eine Maske generiert mit welcher die Daten anschließend maskiert werden. Der korrekte Wert wird nun durch Verschiebung um das Startbit nach hinten erreicht. Für die Option 1 ist dabei nur die Kundennummer interessant. Diese wird überprüft, ob es sich dabei um eine Herstellernummer oder einen Kunden handelt. Handelt es sich um eine Herstellernummer wird ein Master Modus aktiviert, in welchem es möglich ist, vertiefte Einstellungen und Freischaltungen vorzunehmen. Dieser ist bis zum nächsten Neustart aktiv.
Sendefunktionen sind für CAN2 keine vorgesehen, da stand jetzt nur der RFID-Reader auf CAN2 sendet.

Die meisten Daten, die über CAN erhalten werden, sind dafür vorgesehen als visuelle Daten für den Kunden dargestellt zu werden. Die erste Variante zur Darstellung von Daten ist über eine Webseite. Im Folgenden soll daher aufgezeigt werden, wie ein Webserver gehostet wird und welche Funktionalitäten dieser bietet. Der ESP32 bietet zwei Unterschiedliche WLAN-Funktionen. Zum einen kann er als Client betrieben werden sich mit einem WLAN-Netzwerk verbinden. Zum anderen ist er auch in der Lage selbst ein eigenes Netzwerk aufzuspannen und so als Accesspoint für andere WLAN-fähige Geräte zu dienen. Dieser Modus wird für dieses System genutzt. Dafür wird im Setup die Initialisierung des WLAN-Chips durchgeführt. Über die Funktion softAPConfig() aus der WiFi.h-Library wird der ESP32 als Accesspoint gestartet. Die übergebenen Parameter legen dabei die IP-Adresse und alle damit verbundenen Informationen fest. Das dient dazu, dass der Mikrocontroller bei jedem Neustart dieselbe IP-Adresse behält. Neben den IP-Informationen müssen auch die Verbindungsinformationen wie SSID und das Passwort festgelegt werden. Diese werden Initial für jedes Kart identisch im Persistenten Speicher des ESP32 abgelegt, sind aber später individuell änderbar.

```
548  void init_wifi(){
552    // Start Wifi as Accesspoint
554    if(!WiFi.softAPConfig(local_IP, gateway, subnet)) {
555      Serial.println("Fehler beim Setzen der AP-Konfiguration");
556    }
557
558    // Access Point starten
559    if(WiFi.softAP(ESP_storage.getString("WIFI_Name", "SMS REVO SL"),
560      ESP_storage.getString("WIFI_Password"))) {
561
562      // Debugging Infos
563      Serial.println("Access Point gestartet");
564      Serial.print("IP-Adresse des Access Points: ");
565      Serial.println(WiFi.softAPIP());
566    } else {
567      Serial.println("Fehler beim Start des Access Points");
568    }
```

Programmcode 35: Konfigurierung des ESP32 als WLAN-Accesspoint

Damit ist die Konfiguration des WLAN-Moduls abgeschlossen und der Webserver kann initialisiert werden. Für diesen werden die einzelnen Seiten festgelegt, bevor er gestartet werden kann. Jede Seite benötigt dabei eine eigene URL und einen definierten handler, welcher sowohl die graphische Oberfläche als auch die Funktionalität zur Verfügung stellt. Über die Funktion begin() wird der Webserver gestartet. Zusätzlich wurde für zwei Seiten das HTTP-Protokoll definiert.

Als nächstes sollen diese Seiten, deren Aufbau und Funktionen näher betrachtet werden. Dabei soll es weniger um den Aufbau der HTML-Seiten gehen als um die Umsetzung der Funktionen, welche über die Webseiten realisiert werden. Jede Seite besitzt eine Abfrage, ob die Funktionalität des WLAN-Moduls freigegeben ist. Diese erzeugt für jede URL eine Webseite ohne Funktionen. Hintergrund ist, dass der ESP32 immer als Access Point arbeitet und den Webserver hostet, damit es für den Hersteller möglich ist auf Einstellungen zuzugreifen, auch wenn die Funktion für den Kunden deaktiviert ist. Dadurch, dass tatsächlich eine Seite ohne Funktionen erstellt wird, ist es auch nicht möglich über Umwege auf Funktionalitäten, welche eigentlich gesperrt sein sollten zuzugreifen.

```
548   void init_wifi(){
571     kart_server.on("/", handleRoot);
572     kart_server.on("/livedaten", handleLivedaten);
573     kart_server.on("/einstellungen", handleEinstellungen);
574     kart_server.on("/values", handleValues);
575     kart_server.on("/RF_Connect_Return", handleRFConReturn);
576     kart_server.on("/hersteller", HTTP_GET, handleSettingsToggle);   //
577     kart_server.on("/hersteller", HTTP_POST, handleSettingsToggle);  //
578     kart_server.on("/downloadLog", HTTP_GET, handleDownloadLog);
579     kart_server.onNotFound(handleNotFound);
580
581     // Start Webserver
582     kart_server.begin();
583   }
```

Programmcode 36: Festlegungen der URLs für die Webseiten des Webservers

Ist die Funktionalität freigegeben, ist die erste Seite, welche der Kunde betritt, die Root Seite. Diese stellt noch keine Funktion zur Verfügung, welche in den Anforderungen definiert wurde, sondern liefert lediglich eine Übersicht und ein Menü, um die beiden anderen Seiten mit Funktionen zu erreichen. Anhand dieser Seite soll die allgemeine Funktionsweise gezeigt werden. Die Seite wird über HTML aufgebaut. Darüber werden im Abschnitt <style> das Aussehen der Seite und der Elemente auf der Seite definiert. Im Abschnitt <body> werden die Elemente auf der Seite hinzugefügt und mit entsprechenden Funktionen versehen. Im Beispiel der Hauptseite besteht die Seite aus einer Überschrift im Style <h1>, welche den Namen des Karts zeigt. Darunter sind zwei Buttons platziert vom Typ a mit entsprechenden Farben platziert. Die Funktion des Knopfes wird über den Parameter href definiert, indem dort die URL angegeben wird, auf die bei Drücken des Knopfes verlinkt wird. So führt Drücken des Grünen Knopfes auf die Seite /livedaten, der Blaue Knopf verlinkt auf Seite /einstellungen. Die Beschriftung des Knopfes wird hinter dem Link angegeben. Die Darstellung auf der Seite passiert über die Funktion send() des Webserver Objekts. Diese sendet eine Antwort auf eine HTTP-Anfrage des Webservers. Dabei werden der Funktion verschiedene Parameter übergeben. Der erste Parameter ist der HTTP-Status Code, in diesem Fall 200 für OK, danach wird der Typ

des Inhalts angegeben, im Fall dieser Seite wird ein HTML-Text gesendet und schließlich der eigentliche Inhalt der Seite als HTML-Text.

```
975   void handleRoot() {
980     if(ESP_storage.getInt("WiFi_enable", FALSE) == FALSE){
994     else{
995       MAIN_page =
996       "<!DOCTYPE html>"
997       "<html lang=\"de\">"
998       "<head>"
999         "<meta charset=\"UTF-8\">"
1000        "<meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\">
1001        "<title>SMS REVO SL</title>"
1002        "<style>"
1003          "body { margin:0; display:flex; justify-content:center;"
1004          "align-items:center; height:100vh; background:#f2f2f2; "
1005          "font-family:Arial,sans-serif; }"
1006          ".card { background:#fff; padding:20px 30px; border-radius:12px; "
1007          "text-align:center; box-shadow:0 4px 12px rgba(0,0,0,0.15); }"
1008          "h1 { color:red; margin-bottom:20px; }"
1009          "a.btn { display:inline-block; margin:8px; padding:12px 18px; "
1010          "border-radius:8px; text-decoration:none; color:white; "
1011          "font-weight:bold; }"
1012          "a.green { background:green; }"
1013          "a.gray { background:gray; }"
1014          "a.blue { background:blue; }"
1015        "</style>"
1016      "</head>"
1017      "<body>"
1018        "<div class=\"card\">"
1019          "<h1>SMS REVO SL</h1>"
1020          "<a class=\"btn green\" href=\"/livedaten\">Livedaten</a>"
1021          "<a class=\"btn blue\"  href=\"/einstellungen\">Einstellungen</a>"
1022        "</div>"
1023      "</body>"
1024      "</html>";
1025    }
1026
1027    kart_server.send(200, "text/html", MAIN_page);
```

Programmcode 37: HTML-Code zur Darstellung der Webseite /root

Das Verarbeiten des Webservers und das Stellen der HTTP-Anfragen an den Handler, übernimmt der Loop. Dort wird zyklisch die Funktion kart_server.handleClient() aufgerufen, welche nach offenen HTTP-Requests sucht und diese an die Handler weiterreicht, welche für die jeweilige URL definiert wurden.

Die Seite /livedaten, welche über die Hauptseite erreicht werden kann, stellt die Daten vom Kart, welche per CAN an die Erweiterungsplatine gesendet werden, visuell für den Nutzer dar. Im aktuellen Stand der Webseite sind lediglich 3 Werte dargestellt, wie in Abbildung 27 zu sehen, um die Funktionsweise und das Konzept zu zeigen. Die Werte

Abbildung 27: Webseite zur Anzeige der Livedaten

sollen sich automatisch aktualisieren, ohne dass der Kunde dies händisch einleiten muss.
Die Webseite hat dafür ein in den HTML-Text eingefügten Skript-Teil. Dort läuft ein JavaScript Programm, welches alle 200ms die Werte vom ESP32 abfragt und in den Feldern auf der Webseite darstellt. Das Skript beinhaltet eine asynchrone Funktion, welche die Werte von der Seite /values mit der Funktion fetch() abfragt. Bei der Seite /values handelt es sich um eine json-Application, welche die Werte für die darzustellenden Werte über die Funktion response.json zurückgibt.

```
1034  void handleLivedaten() {
1057    else{
1073      "<script>"
1074      "async function updateValues(){"
1075      "  let response = await fetch('/values');"
1076      "  let data = await response.json();"
1077      "  document.getElementById('speed').innerText = data.speed + ' km/h';"
1078      "  document.getElementById('soc').innerText = data.soc + ' %';"
1079      "  document.getElementById('temp').innerText = data.temp + ' °C';"
1080      "}"
1081      "setInterval(updateValues,1000);"
1082      "</script>"
1083      "</head><body>"
1084      "<div class='card'>"
1085      "<h1>Livedaten</h1>"
1086      "<div class='databox'><span class='label'>Geschwindigkeit</span>"
1087        "<span class='value' id='speed'>--</span></div>"
1088
1089      "<div class='databox'><span class='label'>State of Charge</span>"
1090        "<span class='value' id='soc'>--</span></div>"
1091
1092      "<div class='databox'><span class='label'>Akkutemperatur</span>"
1093        "<span class='value' id='temp'>--</span></div>"
1094
1095      "<a class='btn gray' href='/'>Zurück</a>"
1096      "</div>"
1097      "</body></html>";
1098    }
1099    kart_server.send(200, "text/html", LIVEDATEN_page);
```

Programmcode 38: JavaScript Funktion zur Abfrage und Darstellung der Livedaten innerhalb des HTML-Code

Aus diesem json-Datensatz können nun die Werte für Geschwindigkeit, SOC und Temperatur ausgelesen werden und mit der entsprechenden Einheit versehen werden. Diese Werte werden im normalen HTML-Body als Inhalt der Datenboxen dargestellt.

Damit das Auslesen der Werte von der Seite /values funktioniert, muss diese eine json-Application zur Verfügung stellen. Diese liest als erstes die Variablen des ESP32. Dabei handelt es sich um globale Variablen, welche von den Auswertungsfunktionen von CAN beschrieben werden und von allen Funktionen, die diese Daten darstellen wollen, ausgelesen werden können. Diese Daten werden in einen json-String verpackt und als Applikation an der Webserver gesendet.

```
1090
1091    void handleValues() {
1092      float speed, soc, bat_temp;
1093      noInterrupts();
1094      speed = CAN_speed;
1095      soc = CAN_soc;
1096      bat_temp = CAN_battery_temp;
1097      interrupts();
1098
1099      String json = "{";
1100      json += "\"speed\":" + String(speed, 1) + ",";
1101      json += "\"soc\":" + String(soc, 1) + ",";
1102      json += "\"temp\":" + String(bat_temp, 1);
1103      json += "}";
1104
1105      kart_server.send(200, "application/json", json);
1106    }
```

Programmcode 39: Json-App zur Darstellung der Livewerte auf der Webseite

Eine weitere Funktionalität, welche über die WLAN-Schnittstelle und den Webserver geschaffen wird, ist die Änderung von Einstellungen sowohl für den Kunden als auch den Hersteller. Dafür ist die Webseite /einstellungen vorgesehen. Über diese in Abbildung 28 gezeigte Webseite soll es möglich sein alle wichtigen Einstellungen vorzunehmen, ohne dass der Kunde darüber neue Funktionen freischalten kann.

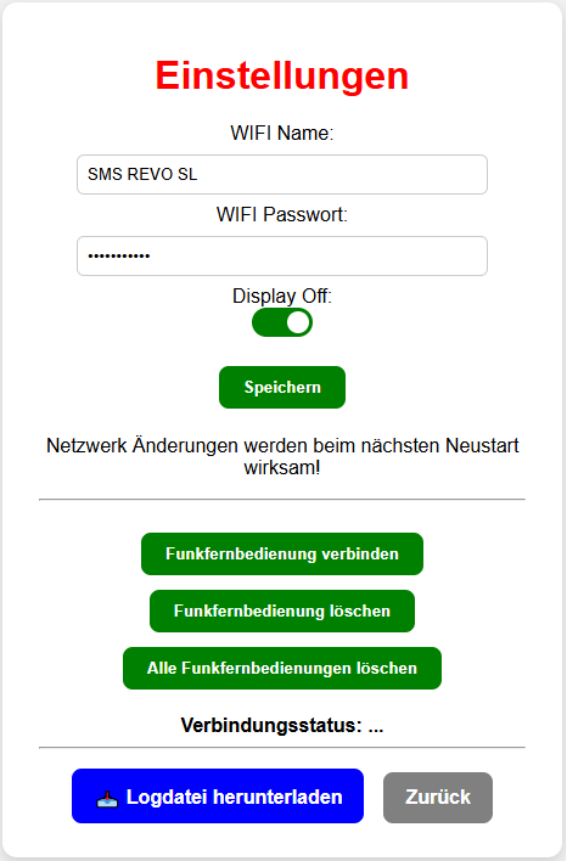

Abbildung 28: Webseite zur Anpassung der Einstellungen des ESP

Dazu gehört zum Beispiel das Festlegen eines WLAN-Namens und eines neuen Passworts. Das ist vor allem für Vereine wichtig, welche mehrere Karts besitzen, um diese anhand des Netzwerknamens auseinanderhalten zu können. Auch das Passwort kann über die Einstellungen geändert werde. So kann jeder Verein sein eigenes Passwort festlegen, um zu verhindern, dass unbefugte mit dem Standardpasswort, welches bei allen Karts voreingestellt ist, auf fremde Karts zugreifen können. Diese Änderungen sollen nicht automatisiert übernommen werden, sondern erst durch Drücken des Speichern Knopfes in den Persistenten Speicher geschrieben werden. Aufgrund der bereits bestehenden WLAN-Verbindung werden diese Änderungen erst nach einem Neustart wirksam. Anders als bei den bisherigen Seiten wird der Aufbau der Seite nicht als HTML-Body, sondern als HTML-Formular realisiert. Dieses sendet einen HTTP-Post, sobald Daten aus dem Formular, wie zum Beispiel die Eingaben für WLAN SSIO und Passwort, vom Client an den Server übertragen werden sollen. Dieser HTTP-Post wird im Mikrocontroller vom Handler abgefragt über die Funktion method(). Wenn in den Feldern Daten eingetragen sind, werden diese im Non-Volatile Storage des ESP32 gespeichert.

```
1108  void handleEinstellungen() {
1112    if(ESP_storage.getInt("WiFi_enable", FALSE) == FALSE){
1131    else{
1132      if (kart_server.method() == HTTP_POST) {
1133        // Werte aus Formular speichern
1134        if (kart_server.hasArg("wifi_name")) {
1135          ESP_storage.putString("WIFI_Name",
1136            kart_server.arg("wifi_name"));
1137        }
1138        if (kart_server.hasArg("wifi_password")) {
1139          ESP_storage.putString("WIFI_Password",
1140            kart_server.arg("wifi_password"));
1141        }
1142
1143        if (kart_server.hasArg("display_off")) {
1144          ESP_storage.putInt("Display_off", true);
1145        } else {
1146          ESP_storage.putInt("Display_off", false);
1147        }
```

Programmcode 40: HTTP-Request Handler für Einstellungen

Dasselbe gilt für den Schalter für Display Off. Dieser soll einen Modus aktivieren, in welchem das Display ausgeht, sobald das Kart losfährt, um zu verhindern, dass es den Fahrer irritiert oder verunsichert. Auch dieser Schalter wird bei einem HTTP-Post ausgelesen und der Wert entsprechend gespeichert. Der HTTP-Post wird dabei durch Drücken des Speicherknopfes gesendet.

Neben der Änderung von Einstellungswerten soll es auch möglich sein Fernbedienungen mit dem Kart zu verbinden. Dafür wurden drei Buttons zum Verbinden und Löschen einzelner Fernbedienungen und dem Löschen aller Fernbedienungen vorgesehen. Die Funktionsweise dieser Knöpfe soll anhand der Verbindung einer Fernbedienung mit dem System erklärt werden. Jeder Knopf wird bei Senden eines HTTP-Post ausgewertet. Über das Übertragene Argument kann der gedrückte Knopf identifiziert werden. Darüber ist nun bekannt, welcher Lern oder Löschmodus aktiviert werden soll. Für den ersten Knopf zum Anlernen einer Fernbedienung ist das Lernmodus 1, für die beiden Löschmodi sind die Modi 5 und 6 vorgesehen. Die Aktivierung des Lernmodus findet dabei über die bereits betrachtete Funktion learn_RFControl() statt.

```
1108  void handleEinstellungen() {
1131    else{
1132      if (kart_server.method() == HTTP_POST) {
1148
1149        // Spezialaktionen über Buttons
1150        if (kart_server.hasArg("action")) {
1151          String act = kart_server.arg("action");
1152          if (act == "remote_bind") {
1153            // Start learning mode for RF Controller
1154            Serial.println("bind Remote Control");
1155            int rv = learn_RFControl(1); // Start learning mode for RF C
1156            if(rv==ERROR){
1157              Serial.println("Pairing Controller failed");
1158              // Write to Website
1159              RF_Connect_Return = "Pairing Controller failed";
1160            }
1161            else{
1162              Serial.println("Pairing Controller successful");
1163              // Write to Website
1164              RF_Connect_Return = "Pairing Controller successful";
1165            }
1166          }
```

Programmcode 41: Handler zur Auswertung der Knöpfe auf der Einstellungen Webseite

Um dem Nutzer eine Rückmeldung über den Status des Verbindungsversuchs zu geben, wird unterhalb der Knöpfe die Variable RF_Connect_Return angezeigt. Tritt, während dem Versuch eines Verbindungsaufbaus, ein Fehler auf, wird diese Variable beschrieben. Ebenso wird darüber ein erfolgreicher Abschluss der Funktion angezeigt. Die Anzeige dieser Variable passiert wieder über ein in den HTML-Code eingefügtes Java Skript. Dieses fragt den Wert von der Webseite /RF_Connect_Return ab und stellt ihn der Seite zur Anzeige zur Verfügung. Dabei handelt es sich bei der URL /RF_Connect_Return nicht wie bei der Livedaten Seite um eine json-Application, sondern hier reicht es aus, den Wert als reinen Text zu übertragen, da es nur um den Wert einer Variablen geht.

Die letzte Funktionalität, welche über diese Webseite realisiert wird, ist der Download der Fehler-Logdatei. Der dafür vorgesehene Button funktioniert genau wie die beiden Knöpfe auf der Hauptseite und verlinkt lediglich zur Webseite /DownloadLog. Diese Seite überprüft automatisch beim Aufruf der Seite, ob die Logdatei vorhanden ist und beginnt den Download, wenn die Datei gefunden wurde. Aktuell wird als Speicherort für die Logdatei der Persistente Speicher des Mikrocontrollers genutzt. Zukünftig soll dafür der auf der Platine vorgesehene Flashspeicher genutzt werden.

```
1905  void handleDownloadLog() {
1906    if (SPIFFS.exists("/Error.log")) {
1907      File file = SPIFFS.open("/Error.log", "r");
1908      if (file) {
1909        // Header setzen, damit der Browser einen Download startet
1910        kart_server.sendHeader("Content-Type", "text/plain");
1911        kart_server.sendHeader("Content-Disposition",
1912          "attachment; filename=Error.log");
1913
1914        kart_server.sendHeader("Connection", "close");
1915        kart_server.streamFile(file, "text/plain");
1916        file.close();
1917        return;
1918      }
1919    }
```

Programmcode 42: Übertragung der Logdatei als Download auf den Verbundenen PC per Netzwerkschnittstelle

Die Eingabe der gerade angesprochenen Einstellungen findet dabei über die Seite /hersteller statt. Hierrüber soll es für den Hersteller möglich sein alle Hauptfunktionen der Option 1 zu aktivieren oder zu deaktivieren. Daher darf es nicht möglich sein, dass Kunden diesen Bereich erreichen können. Zu diesem Zweck wird der Master Modus genutzt, welcher nur über die Verwendung der RFID-Karte mit Herstellerkundennummer erreicht werden kann. Daher ist diese Seite auch die einzige, welche immer aktiv ist, selbst wenn die Funktionalität der WLAN-Schnittstelle für den Kunden deaktiviert wurde.

```
1735  void handleSettingsToggle() {
1739    if(Master_Mode_active == true){
1740      // AJAX-POST-Handler: nur speichern, keine HTML-Seite
1741      if (kart_server.method() == HTTP_POST) {
1742        if (kart_server.hasArg("key") && kart_server.hasArg("state")) {
1743          String key = kart_server.arg("key");
1744          String state = kart_server.arg("state");
1745          bool value = (state == "true");
1746
1747          if (key == "Display_enable") {
1748            ESP_storage.putInt("Display_enable", value);
1749            Serial.printf("Display_enable -> %s\n", value ? "true" : "fa
1750          }
1751          if (key == "RF_enable") {
1752            ESP_storage.putInt("RF_enable", value);
1753            Serial.printf("RF_enable -> %s\n", value ? "true" : "false")
1754          }
1755 >        if (key == "WiFi_enable") {···
1759 >        if (key == "RFID_enable") {···
1763 >        if (key == "LED_enable") {···
1767 >        if (key == "RF_CAN_en") {···
```

Programmcode 43: AJAX Post Handler für Schieberegler auf Hersteller Webseite

Für jede große Einstellung ist ein Schiebebutton wie auf der /einstellungen-Seite vorgesehen, welcher zusätzlich die aktuelle Einstellung anzeigt. Anders als auf der vorherigen Seite, sollen die Werte aber gespeichert werden, sobald der Knopf verändert wird. Dafür wird ein AJAX (Asynchronous JavaScript And XML) Post Request verwendet. Damit findet die Datenaustausch wie bei einem normale Post Request statt, nur mit dem Unterschied, dass die Seite nicht neu geladen werden muss. Der Ablauf dieses Post Request und das Speichern der Daten soll im Code anhand zweier Einstellungen gezeigt werden. Der Post Request übergibt dabei zwei Argumente. Key ist der Name der Einstellung, beispielsweise Display_Enable, zur Freischaltung der Display-Funktion. State ist der Wert des Schalters. Dieser Wert wird nun der entsprechenden Einstellung übergeben und im NVS des ESP32 gespeichert.

Die Darstellung der Schalter übernimmt wie bei den vorherigen Seiten ein Formular. Dabei wird die Anzeige im Beispiel über die Variable rfEnable sichergestellt. Diese Anzeigeoption sorgt dafür, dass der Schalter Grün hinterlegt wird, sobald der Wert im Speicher des ESP32 auf True gesetzt wird. Diese Variablen werden dafür bei jedem laden der Webseite auf den aktuellen Wert des Speichers gesetzt.

```
1819    // RF_enable Switch
1820    "<div class='label'>Funkempfänger</div>"
1821    "<label class='switch'>"
1822    "<input type='checkbox' id='rfToggle' " + String(rfEnable ? "checked"
1823    "<span class='slider'></span>"
1824    "</label>"
```

Programmcode 44: Darstellung eines Schiebeschalters im HTML-Code

Für die Funktionalität des Speicherns wird auch hier wieder ein JavaScript im HTML-Code eingesetzt. Dieses besteht aus zwei Abschnitten. Zum einen wird hier die Funktion sendToggle() realisiert, welche den HTTP-Post mit den entsprechenden Argumenten an den Server sendet. Zum anderen wird für jeden Schalter ein Event Listener hinzugefügt, welcher auf Änderungen des jeweiligen Schalters wartet. Entdeckt einer dieser Listener eine Veränderung bei seinem Schalter, ruft er die Funktion sendToggle mit den passenden Argumenten auf und sendet damit den HTTP-Post für den jeweiligen Schalter an den Server.

```
1868        "<script>"
1869        "function sendToggle(key, state){"
1870        "  fetch('/hersteller',{method:'POST',headers:{'Content-Type':"
1871        "     'application/x-www-form-urlencoded'},"
1872        "  body:'key='+key+'&state='+(state?'true':'false')});"
1873        "}"
1874        "document.getElementById('rfToggle').addEventListener('change',function(){"
1875        "  sendToggle('RF_enable',this.checked);"
1876        "});"
```

Programmcode 45: JavaScript Funktion zur Einrichtung des Event Listeners für die Knöpfe der Webseite /hersteller

Damit die Schalter ihre Einstellungen speichern können, muss dieser Speicher vorher definiert und die entsprechenden Variablen angelegt werden. Dafür wird im Setup die Funktion init_storage() aufgerufen. Diese Funktion definiert den Speicherbereich für die Einstellungen und ermöglicht den Zugriff auf den Speicher über ein Objekt. Das passiert über die Funktion begin(). Diese Initialisiert den Speicherbereich „settings“. Durch den Parameter False wird er als beschreibbar angelegt. Sollte der Speicher bereits existieren, wird der Speicherbereich geöffnet.

```
465  void init_storage(){
469    // Save settings
470    ESP_storage.begin("settings", false);
471
472    if(ESP_storage.getInt("RF_enable", -1) == -1){
473      // Variable is not stored at the moment
474      // Init Variable
475      ESP_storage.putInt("RF_enable", true);
476    }
477    if(ESP_storage.getInt("Display_enable", -1) == -1){
478      // Variable is not stored at the moment
479      // Init Variable
480      ESP_storage.putInt("Display_enable", true);
481    }
482 >  if(ESP_storage.getInt("WiFi_enable", -1) == -1){ ···
487 >  if(ESP_storage.getInt("RFID_enable", -1) == -1){ ···
492 >  if(ESP_storage.getInt("LED_enable", -1) == -1){ ···
497 >  if(ESP_storage.getInt("RF_CAN_en", -1) == -1){ ···
502 >  if(ESP_storage.getInt("Display_off", -1) == -1){ ···
507 >  if(ESP_storage.getString("WIFI_Name", "unknown") == "unknown"){ ···
512 >  if(ESP_storage.getString("WIFI_Password", "unknown") == "unknown"){
```

Programmcode 46: Initialisierung des Persistenten Speichers im ESP32

Danach legt die Funktion alle benötigten Variablen im Speicher an. Das wird über die Funktion put...() mit dem entsprechenden Datentyp erreicht, welcher angelegt werden soll. Der Funktion wird dabei ein Key-Value Paar übergeben. Der Wert wird so abgespeichert, dass er über den Schlüssel wieder ausgelesen werden kann. Damit die Variable nicht überschrieben wird, sollte sie bereits existieren, wird vorher aus dem Speicherbereich gelesen. Das passiert über die Funktion get...(), in Kombination mit dem auszulesenden Datentyp. Dabei wird der Key definiert, unter welchem die Variable abgespeichert wurde. Der zusätzlich angegebene Wert ist der Rückgabewert, sollte die Variable im Speicher noch nicht existieren. Die Variable soll also nur beschrieben werden, wenn der Rückgabewert der Funktion dem Wert -1 entspricht. Da es sich bei den zu speichernden Variablen um boolesche Werte handelt, kann der gespeicherte Wert nie den Wert -1 annehmen, außer die Variable existiert noch nicht. Auf diese Weise werden für folgende Einstellungen Variablen angelegt:

RF_enable	Aktivierung für Funkempfänger Funktionen
Display_enable	Aktivierung für Display Funktionen
WiFi_enable	Aktivierung für WiFi Funktionen
RFID_enable	Aktivierung für RFID-Funktionen
LED_enable	Aktivierung für Status-LED Funktionen
RF_CAN_en	Aktivierung für Funksignale via CAN (Kompatibilität)
Display_off	Aktivierung Display Aus Modus beim Fahren
WIFI_Name	Netzwerkname für WLAN
WIFI_Password	Netzwerkpasswort für WLAN

Tabelle 9: Variablen im Persistenten Speicher mit ihrer Anwendung

Alle diese Variablen sollten eigentlich im Externen Flashspeicher, welcher auf der Platine platziert ist, gespeichert werden. Dieser hat den Vorteil, dass die Variablen auch nach dem Flashen eines neuen Softwarestands erhalten bleiben. Außerdem wird aktuell das WLAN-Passwort im Klartext gespeichert, sodass es nicht besonders sicher ist. Im externen Flash wäre der Speicherbereich verschlüsselt, sodass es nicht so einfach wäre, unbefugt an das Passwort zu gelangen. Auch die Logdatei soll zukünftig auf dem Externen Flash gespeichert werden. Leider ist der Chip aktuell nur mit großen Lieferzeiten bestellbar, weswegen er auf der ersten bestellten Version der Platine nicht bestückt ist. Um die allgemeine Funktion trotzdem testen zu können, wird statt dem Flash der NVS verwendet.

Ähnliches gilt für das Display. Aufgrund der Langen Lieferzeit für die Platine und das Display war keine Zeit mehr vorhanden die Funktionen des Displays zu testen und einzelne Graphische Seiten aufzubauen, welche die Daten entsprechend aufbereiten und diese zu testen. Auf die Funktionalität des Displays wird deshalb softwareseitig vorläufig verzichtet, da alle Relevanten Informationen auch über das Webinterface zur Verfügung gestellt werden können.