

Team 03 - Paketdrohne

Projektbericht

M. Prenzlow, J. Reichmann, T. Tian, H. Zhang

Embedded Systems
Wintersemester 2023/24

Inhaltsverzeichnis

1	Projektbeschreibung	1
2	SA/RT Modell	1
2.1	Kontextdiagramm	1
2.2	Datenflussdiagramm	2
2.3	Zustandsübergangsdiagramm	2
2.4	Datenverzeichnis	3
2.5	Minispezifikation	3
2.6	Prozessaktivierungstabelle	5
3	Hardware	6
3.1	LED-Matrix	7
3.2	Nachbar-Pins	7
4	Software	8
4.1	LED-Ansteuerung	8
4.1.1	WS2812 DMA Library	8
4.1.2	WS2812 SPI Library	9
4.2	Labyrinth	10
4.3	Pseudozufallszahlengenerator (PRNG)	10
4.3.1	Implementierung des PRNG	11
4.3.2	Nutzung der Zahlenliste aus <code>numbers.c</code>	11
4.4	Generierung des Labyrinths	11
4.4.1	Strukturen in <code>maze.h</code>	11
4.4.2	Der Generierungsprozess	12
4.4.3	Schlüsselkonzepte	12
4.5	Lösung des Labyrinths	12
4.5.1	Der Lösungsalgorithmus	13
4.5.2	Schlüsselkonzepte	13
4.5.3	Implementierungsdetails	13
4.5.4	Animation der Drohne auf der LED-Matrix	13
5	Tests	15
5.1	Testaufbau	15
5.2	Team 13 (Lego)	15
5.3	Team 13 (Lego) und Team 12 (Roboterarm2)	16

1 Projektbeschreibung

In den Projekten im Rahmen des Moduls „Embedded Systems“ wurde eine Weiterleitung von Paketen zwischen mehreren Projektteams simuliert. Die Basis bildete ein Master-Slave-Protokollstack (MMCP-Protokoll), der auf NUCLEO-F401RE Mikrocontrollerboards implementiert wurde. Zur Kommunikation wurden die Mikrocontroller-Boards auf einem sogenannten Baseboard zusammengeschaltet, wodurch die verschiedenen Teams in einer Kette über RS-232 mit dem Master kommunizieren konnten. Weitere Details zum MMCP-Protokoll und zum Baseboard finden sich im Skript des Moduls.

Die Paketweiterleitung wurde von jedem Team individuell als eine Applikation auf dem MMCP-Protokoll nach dem SA/RT-Entwurfsprinzip realisiert. In diesem Projekt wurde zur Visualisierung der Paketübergabe eine Paketdrohne auf einer LED-Matrix simuliert, welche ihren Weg durch ein zufällig generiertes Labyrinth zum Übergabepunkt des Nachbarteams findet.

In diesem Projektbericht wird zunächst das vollständige SA/RT-Modell vorgestellt, dann die Hardware erläutert und anschließend auf die Software für die LED-Ansteuerung und das Labyrinth eingegangen. Abschließend werden Tests des Projektes mit den Nachbarteams beschrieben.

2 SA/RT Modell

2.1 Kontextdiagramm

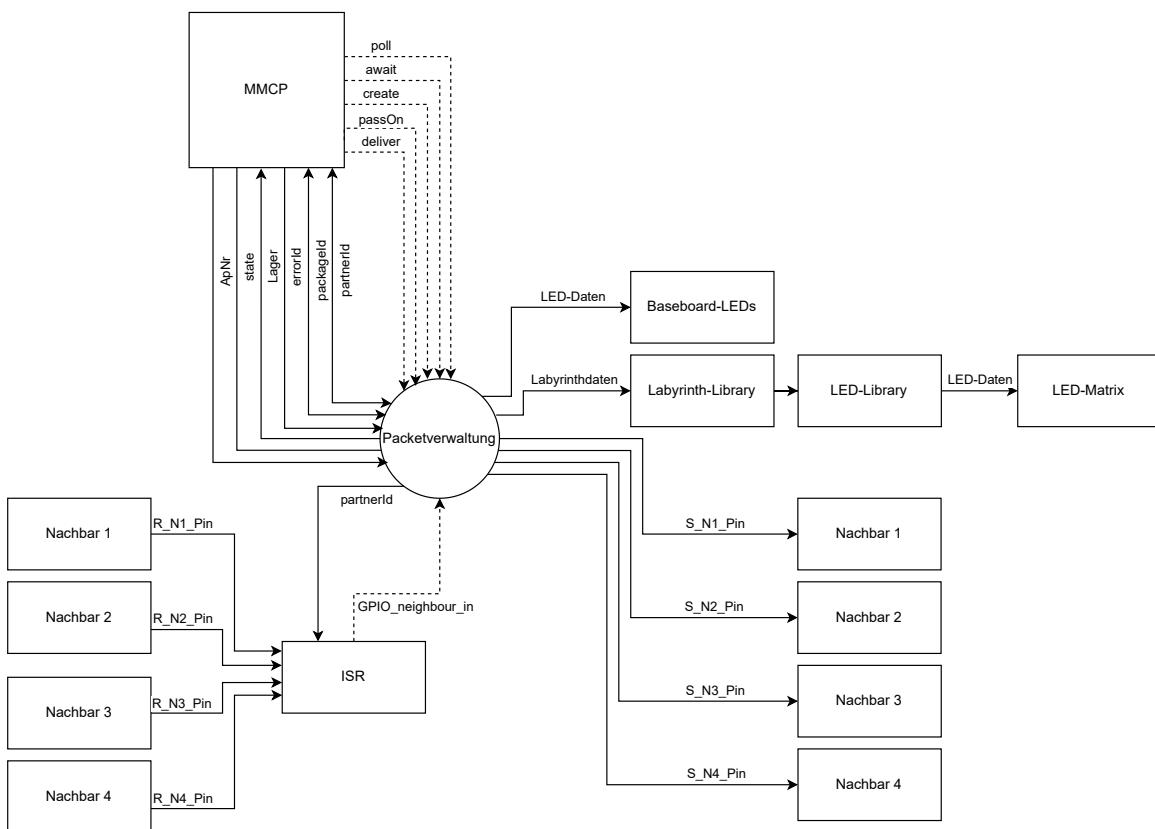


Abbildung 1: Kontextdiagramm

2.2 Datenflussdiagramm

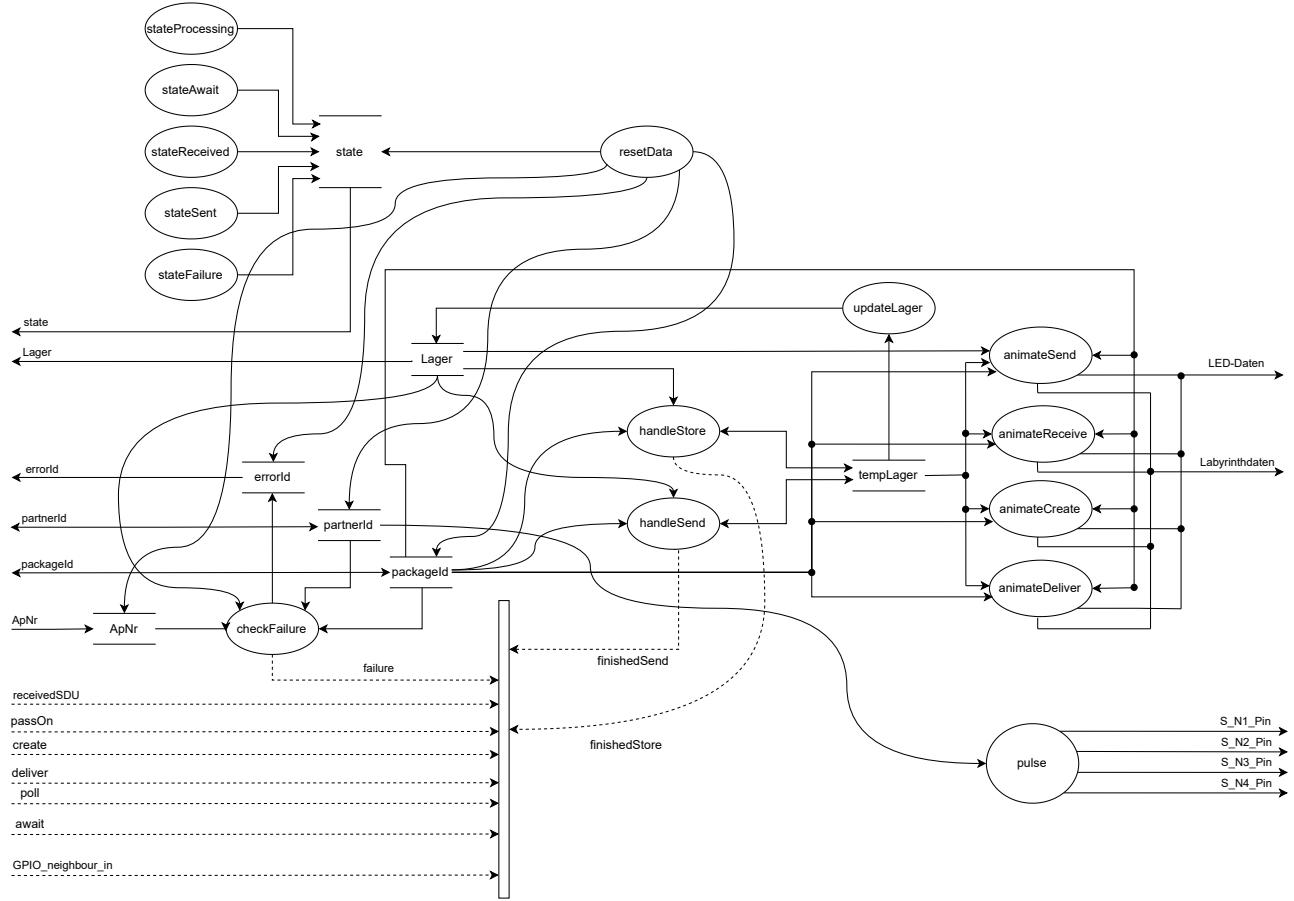


Abbildung 2: Datenflussdiagramm

2.3 Zustandsübergangsdiagramm

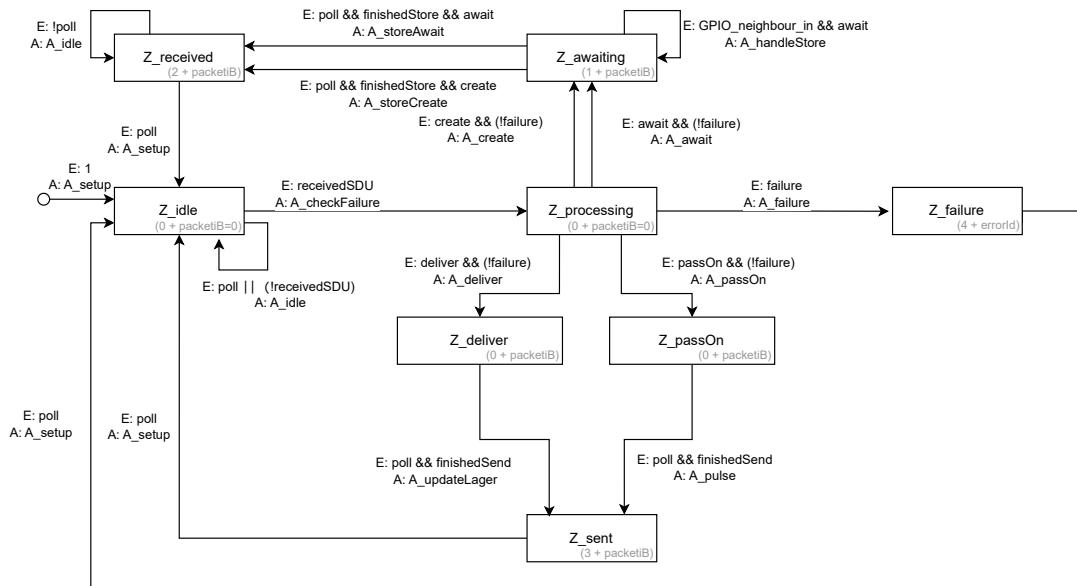


Abbildung 3: Zustandsübergangsdiagramm

2.4 Datenverzeichnis

```

DATA state = " uint_8 "
DATA errorId = " uint_8 "
DATA partnerId = " uint_8 "
DATA packageId = " uint_8 "
DATA Lager = " uint_8[6] "
DATA tempLager = " uint_8[6] "
DATA ApNr = " uint_8 "

DATA receive = " bool " * Kontrollfluss *
DATA passOn = " bool " * Kontrollfluss *
DATA create = " bool " * Kontrollfluss *
DATA deliver = " bool " * Kontrollfluss *
DATA poll = " bool " * Kontrollfluss *
DATA await = " bool " * Kontrollfluss *
DATA finishedSend = " bool " * Kontrollfluss *
DATA finishedStore = " bool " * Kontrollfluss *
DATA failure = " bool " * Kontrollfluss *
DATA receivedSDU = " bool " * Kontrollfluss *
DATA GPIO_neighbour_in = " bool " * Kontrollfluss *

```

2.5 Minispezifikation

stateProcessing

Setzt **state** auf 0. Dieser Code steht für den Status „processing“, welcher dem Master über das MMCP-Protokoll zurückgemeldet wird.

stateAwait

Setzt **state** auf 1. Dieser Code steht für den Status „awaiting“, welcher dem Master über das MMCP-Protokoll zurückgemeldet wird.

stateReceived

Setzt **state** auf 2. Dieser Code steht für den Status „received“, welcher dem Master über das MMCP-Protokoll zurückgemeldet wird.

stateSent

Setzt **state** auf 3. Dieser Code steht für den Status „sent“, welcher dem Master über das MMCP-Protokoll zurückgemeldet wird.

stateFailure

Setzt **state** auf 4. Dieser Code steht für den Status „failure“, welcher dem Master über das MMCP-Protokoll zurückgemeldet wird.

handleStore

Speichert Paket mit der Nummer **packageId** im **tempLager** Array an der ersten freien Stelle **i**, wo **tempLager[i] == 0** gilt.

handleSend

Löscht Paket mit der Nummer `packageId` aus dem `tempLager` Array, setzt also an der Stelle `i` des Pakets `tempLager[i] = 0`.

updateLager

Kopiert den `tempLager` Array zum `Lager` Array.

animateSend

Senden eines Paketes auf der LED-Anzeige des Baseboards animieren und den Inhalt von `Lager` mit vorgeschriebenen Farben für jede verschiedene Paketnummer `packageId` darstellen. Ruft Funktionen der Labyrinth-Library auf, welche das Labyrinth generieren, lösen und anschließend den Weg der Paketdrohne durch das Labyrinth animieren. Genaue Beschreibung der Funktionen in 4.2.

animateReceive

Empfangen eines Paketes auf der LED-Anzeige des Baseboards animieren und den Inhalt von `Lager` mit vorgeschriebenen Farben für jede verschiedene Paketnummer `packageId` darstellen. Ruft Funktionen der Labyrinth-Library auf, welche das Labyrinth generieren, lösen und anschließend den Weg der Paketdrohne durch das Labyrinth animieren. Genaue Beschreibung der Funktionen in 4.2.

animateCreate

Erstellen eines Paketes auf der LED-Anzeige des Baseboards animieren und den Inhalt von `Lager` mit vorgeschriebenen Farben für jede verschiedene Paketnummer `packageId` darstellen.

animateDeliver

Löschen eines Paketes auf der LED-Anzeige animieren des Baseboards und den Inhalt von `Lager` mit vorgeschriebenen Farben für jede verschiedene Paketnummer `packageId` darstellen.

pulse

Gibt eine steigende Flanke mit der Dauer 1 ms auf dem zu `partnerId` korrespondierenden GPIO-Pin aus. Die Zuordnung von Pins und Nachbarn ist als Konstante im Code realisiert.

checkFailure

Überprüft den empfangenen Befehl auf logische Fehler und Ausführbarkeit.

Zuerst wird überprüft, ob `partnerId` ein bekannter Nachbar bzw. 0 ist. Ist dies nicht der Fall, so wird `errorId = 4` und `failure = TRUE` gesetzt. Die Liste der Nachbarn ist als Konstante im Code realisiert.

Ist `ApNr == 42`, ein Paket soll also erstellt oder empfangen werden, wird überprüft, ob der `Lager` Array noch einen freien Platz `Lager[i] == 0` hat. Ist dies nicht der Fall, so wird `errorId = 2` und `failure = TRUE` gesetzt.

Zudem wird überprüft, ob `packageId` bereits im `Lager` Array vorhanden ist. Ist dies der Fall, so wird `errorId = 1` und `failure = TRUE` gesetzt.

Ist `ApNr == 43`, ein Paket soll gelöscht oder weitergeleitet werden, wird überprüft, ob der `packageId` im `Lager` Array existiert. Ist dies nicht der Fall, so wird `errorId = 3` und `failure = TRUE` gesetzt. Zuletzt wird überprüft, ob `packageId` eine gültige Paketnummer (zwischen einschließlich 1 und 16) ist. Ist dies nicht der Fall, so wird `errorId = 5` und `failure = TRUE` gesetzt.

resetData

Setzt packageId, partnerId, errorId und ApNr auf 0. Setzt receive, passOn, create, deliver, poll, await, failure, finishedStore, finishedSend, receivedSDU auf FALSE.

2.6 Prozessaktivierungstabelle

Aktion/Prozess	stateProcessing	stateAwait	stateReceived	stateSent	stateReceived	stateFailure	handleStore	handleSend	updateLager	animateSend	animateReceive	animateCreate	animateDeliver	pulse	checkFailure	resetData
A_setup	2															1
A_deliver	1								2							
A_passOn	1								2							
A_failure					1											
A_pulse			4						3	1				2		
A_updateLager			3						2				1			
A_await	1															
A_create	1					2										
A_handleStore			3			1										
A_storeAwait				3					2		1					
A_storeCreate					3				2		1					
A_checkFailure	2														1	
A_idle																

Tabelle 1: Prozessaktivierungstabelle

3 Hardware

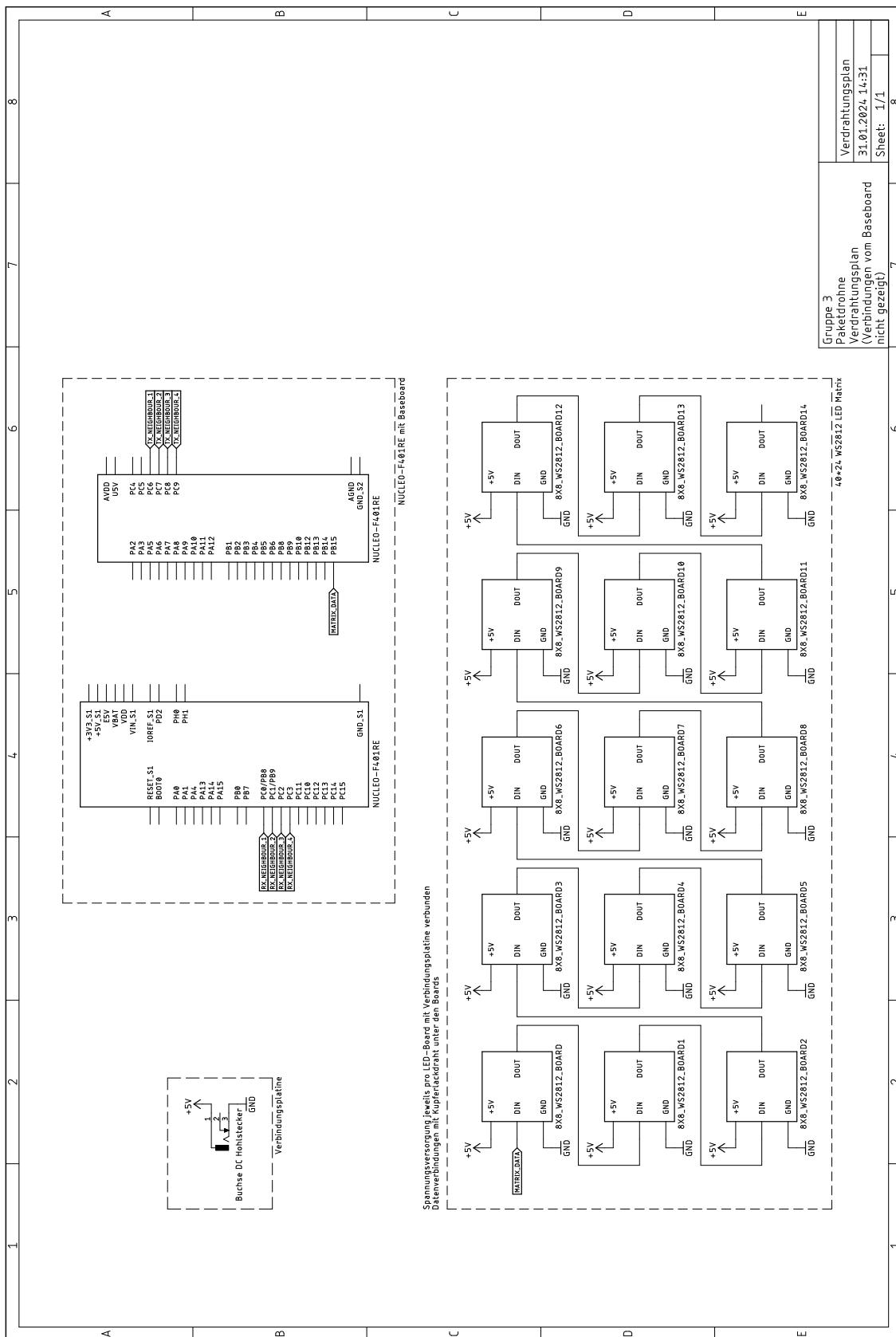


Abbildung 4: Verdrahtungsplan

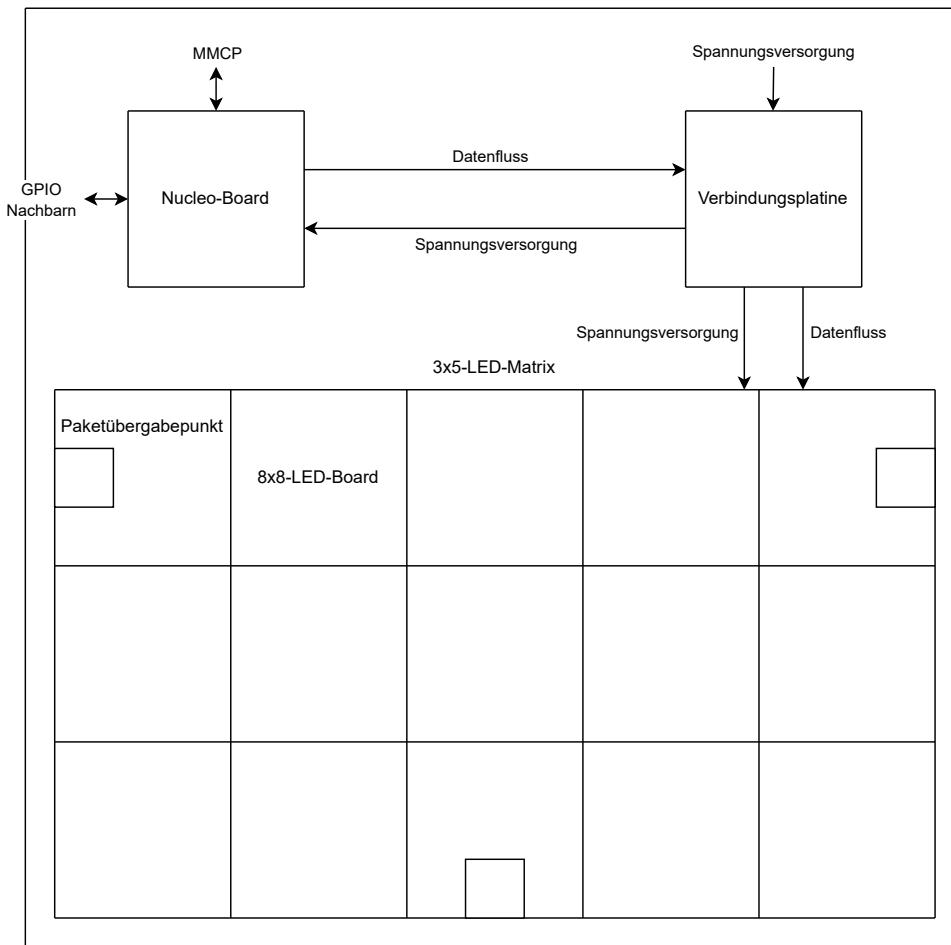


Abbildung 5: Konstruktionsskizze

3.1 LED-Matrix

Wie im Verdrahtungsplan (Abb. 4) gezeigt, besteht die 40x24 LED-Matrix aus 15 einzelnen Boards, welche jeweils 8x8 LEDs des Typs WS2812 besitzen. Alle LED-Boards sind mit Kupferlackdraht in Reihe geschaltet, sodass die gesamte Matrix mit einer Datenleitung angesteuert werden kann. Zudem hat jedes einzelne Board für die Spannungsversorgung zwei Leitungen zur Verbindungsplatine, um einem Spannungsfall in der Kette entgegenzuwirken.

3.2 Nachbar-Pins

Es können maximal 4 Nachbarn angeschlossen werden. Dafür gibt es für jeden Nachbar einen Empfangs- und einen Sendepin (**RX_Neighbour_x** bzw. **TX_Neighbour_x**). Die Empfangspins sind mit einem internen Pull-Down Widerstand konfiguriert. Um Interferenzen entgegenzuwirken, wurden alle Verbindungsleitungen mit einer nach Masse verbundenen Leitung umwickelt.

4 Software

4.1 LED-Ansteuerung

Die LEDs des Baseboards werden wie im Skript beschrieben mit einem PWM Timer über DMA angesteuert, um den Prozessor zu entlasten. Mit dieser Methode müssen allerdings im DMA-Buffer pro LED 24 32-Bit Integers gespeichert werden. Demnach ist eine Ansteuerung der Matrix mit 960 LEDs aufgrund des begrenzten RAM des Controlles so nicht möglich. Deshalb wurde zur Ansteuerung die SPI-Peripherie des µC verwendet.

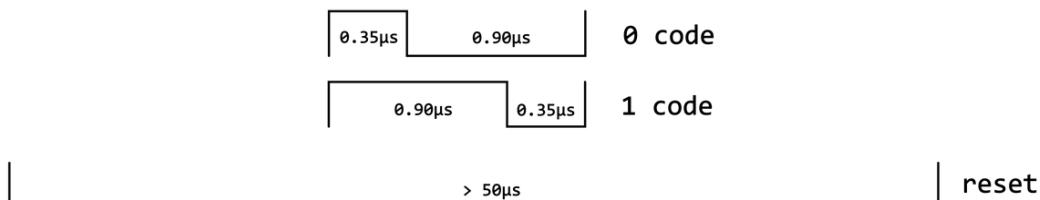


Abbildung 6: Timings für die WS2812 LEDs [ws2812_SPI]

Um die timings für die LEDs zu generieren (Abb. 6), wurde mit 8 SPI-Bits pro Bit für das WS2812-Protokoll eine “0” als 10000000 und eine “1” als 11111100 codiert. Mit einem Prescaler für die SPI-Frequenz von 16 und einer Taktrate von 84 MHz ergibt sich eine Bitrate von 5,25 Mb/s. Da die LEDs sehr tolerant gegenüber Unterschieden im timing sind, funktionieren sie von ca. 3 bis 6 Mb/s. Durch das Nutzen des SPI-Busses mit 8 Bit pro WS2812-Bit wurde so die RAM-Auslastung auf 1/4 gegenüber der PWM-Methode verkleinert. Die Daten werden per DMA an die SPI-Peripherie übermittelt. Der Buffer wird zyklisch an die LEDs gesendet, sodass lediglich in den Buffer geschrieben werden muss und keine Aufrufen einer Funktion zur Aktualisierung der LEDs nötig ist.

4.1.1 WS2812 DMA Library

Die im Skript beschriebene Ansteuerung der LEDs des Baseboards wurde in einer Library realisiert.

Konstanten:

`WS2812_DMA_*`: Verschiedene Konstanten, die die Anzahl von LEDs, Buffergrößen und die timings für die WS2812 LEDs konfigurieren.

Typen:

`PixelRGB_t`: Eine union, die ein RGB-Pixel mit drei 8-Bit Farbkomponenten oder einen einzelnen 32-Bit Wert, für einfache Farbänderung enthält.

```

1 typedef union
2 {
3     struct
4     {
5         uint8_t b; /* blue */
6         uint8_t r; /* red */
7         uint8_t g; /* green */
8     } c; /* color */
9     uint32_t data; /* 32-bit for alignment */
10 } PixelRGB_t;

```

Globale Variablen

`WS2812_DMA_HANDLE`: Die `TIM_HandleTypeDef` für den PWM timer.

`ws2812_DMA_buffer`: DMA Buffer

`ws2812_DMA_buffer_ptr`: Pointer zum DMA Buffer

`ws2812_DMA_pixels`: Ein Array vom Typ `PixelRGB_t`, welcher die Farbwerte von jeder LED enthält.

Funktionen:

`void ws2812_DMA_init(void)`: Initialisiert den DMA Buffer mit dem Wert 0 (Schwarz) für jede LED.

`void ws2812_DMA_write(PixelRGB_t* pixel)`: Schreibt Farbdaten für einen einzelnen Pixel in den DMA Buffer.

`void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef *htim)`: Callbackfunktion, welche aufgerufen wird, wenn alle Daten aus dem DMA Buffer per PWM ausgegeben wurden. Stoppt den PWM timer.

`void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)`: Callbackfunktion für einen Timer zur Animation der LEDs.

4.1.2 WS2812 SPI Library

Die in 4.1 beschriebene Ansteuerung der LEDs der Matrix wurde in einer Library realisiert.

Konstanten:

`WS2812_SPI_*`: Verschiedene Konstanten, die die Anzahl von LEDs, Buffergrößen und die timings für die WS2812 LEDs konfigurieren.

`WS2812_SPI_FILL_BUFFER`: Ein Macro, welches den SPI Buffer mit den Farbdaten für die WS2812 LEDs füllt.

Typen:

`PixelRGB_t`: Eine union, die ein RGB-Pixel mit drei 8-Bit Farbkomponenten oder einen einzelnen 32-Bit Wert, für einfache Farbänderung enthält.

Globale Variablen

`WS2812_SPI_HANDLE`: Die `SPI_HandleTypeDef` der SPI Peripherie.

`ws2812_SPI_buffer`: Der Buffer für die SPI Daten.

Funktionen:

`void ws2812_SPI_init(void)`: Initialisiert den DMA Buffer mit dem Wert 0 (Schwarz) für jede LED und startet die SPI DMA Übertragung.

`void ws2812_SPI_pixel(uint8_t x, uint8_t y, PixelRGB_t* color)`: Setzt den Farbwert eines Pixels an den entsprechenden Koordinaten.

`void ws2812_SPI_pixel_all(PixelRGB_t* color)`: Setzt alle Pixel auf den gegebenen Farbwert.

`void ws2812_SPI_draw(PixelRGB_t** picture, uint8_t width, uint8_t height):` Setzt die LEDs auf die im zweidimensionalen Array `picture` gepseicherten Farbwerte.

`void HAL_SPI_TxCpltCallback(SPI_HandleTypeDefDef* hspi):` Callbackfunktion, welche aufgerufen wird, wenn alle Daten aus dem DMA Buffer über SPI ausgegeben wurden. Startet die SPI DMA Übertragung erneut.

Verwendung des Lookup Tables in der LED-Ansteuerung

Ein wesentliches Element der Ansteuerung der LEDs der Matrix ist der Einsatz eines Lookup Tables, der in `lookup_table.c` definiert und initialisiert wird. Der Lookup-Table bildet Koordinaten (x, y) auf einen linearen Index ab, der den Positionen der LEDs in der seriellen Anordnung in der Matrix entspricht.

```
1 typedef struct {
2     uint16_t rows;
3     uint16_t cols;
4     uint16_t** index;
5 } LookupTable;
```

Innerhalb von `ws2812_SPI.c` wird der Lookup-Table genutzt, um die Position jeder LED in der Matrix zu bestimmen und die entsprechenden Farbdaten korrekt im SPI-Datenpuffer zu positionieren. Dies ermöglicht es, spezifische LEDs basierend auf ihren Koordinaten anzusteuern und komplexe Muster oder Animationen auf der LED-Matrix zu erzeugen.

```
1 void ws2812_SPI_pixel(uint8_t x, uint8_t y, PixelRGB_t* color) {
2     uint8_t* ptr = &ws2812_SPI_buffer[24 * lookupTable.index[y][x]];
3     // Setze Farbdaten im Buffer...
4 }
```

4.2 Labyrinth

In diesem Abschnitt wird eine Software zur Generierung und Lösung von Labyrinthen auf einer LED-Matrix vorgestellt. Die Matrix besteht aus 40x24 WS2812 LEDs. Es wird eine Paketdrohne simuliert, die Pakete innerhalb eines virtuellen Labyrinths transportiert. Das Labyrinth wird dabei dynamisch generiert und der Start- und Zielpunkt hängt von der Lage des Nachbarteams und der Zustellrichtung ab. Auf der Nordseite der Matrix befindet sich das Nucleoboard mit Lager sowie Inbound- und Outbound-Punkten. In den anderen Himmelsrichtungen können sich Nachbarteams befinden, von denen Pakete empfangen oder an die Pakete gesendet werden können.

Das Ziel der Software ist es, den Pfad der Paketdrohne durch das Labyrinth zu berechnen und zu visualisieren. Dabei werden die Drohne und das transportierte Paket auf der LED-Matrix animiert dargestellt. Die Drohne folgt einem vorgegebenen Pfad vom Startpunkt zum Zielpunkt, um das Paket effizient zu liefern. Die Farbe des Pakets wird dabei durch eine vorgegebene Paketfarbe repräsentiert, die im Verlauf der Animation sichtbar ist.

Die Software besteht aus mehreren Komponenten, wie der Generierung des Labyrinths, der Lösung des Labyrinths mittels eines Pfadsuchalgorithmus und der Darstellung der Drohnenbewegung auf der LED-Matrix. Diese Dokumentation beschreibt die Schlüsselfunktionen der einzelnen Quelldateien und erläutert die zugrundeliegenden Konzepte und Algorithmen.

4.3 Pseudozufallszahlengenerator (PRNG)

Der Pseudozufallszahlengenerator, implementiert in `prng.c`, spielt eine zentrale Rolle in der Generierung von Labyrinthen und der Entscheidungsfindung innerhalb der Software. Er nutzt eine vordefinierte Liste von Zahlen aus `numbers.c`, um eine Sequenz von scheinbar zufälligen Werten zu erzeugen, die dennoch reproduzierbar und vorhersagbar sind.

4.3.1 Implementierung des PRNG

Die Datei `prng.c` definiert einen Pseudozufallszahlengenerator mithilfe einer Struktur, die einen Zeiger auf ein Array von Zahlen, einen Index und die Größe des Arrays enthält. Diese Struktur ermöglicht es, durch die Zahlenliste zu iterieren und „zufällige“ Werte für die Labyrinthgenerierung und -navigation zu liefern.

```

1 typedef struct {
2     int* num; /* Zeiger auf Array von vordefinierten Zahlen */
3     int ind; /* Aktueller Index im Array */
4     int size; /* Groesse des Arrays */
5 } PseudoRNG;

```

Die Initialisierung des PRNG erfolgt durch die Funktion `initPRNG`, die das Array von Zahlen, seinen Index und die Größe setzt. Die Funktion `getRand` wird verwendet, um den nächsten Wert aus dem Array zu extrahieren, wobei der Index nach jeder Operation inkrementiert wird.

4.3.2 Nutzung der Zahlenliste aus `numbers.c`

Die Zahlenliste in `numbers.c` enthält eine statisch definierte Sequenz von Werten, die als Grundlage für den PRNG dienen. Diese Liste ermöglicht eine effiziente Umsetzung eines pseudozufälligen Zahlengenerators.

```

1 int numbers[] = {57341, 32796, ..., 29396};

```

4.4 Generierung des Labyrinths

Die Generierung des Labyrinths wird hauptsächlich durch die Funktionen innerhalb der Datei `maze.c` gesteuert, unterstützt durch globale Definitionen und Hilfsfunktionen aus anderen Quelldateien wie `numbers.c` und `prng.c` für die Pseudo-Zufallszahlengenerierung.

4.4.1 Strukturen in `maze.h`

Für die Generierung und Handhabung des Labyrinths definiert die Datei `maze.h` mehrere zentrale Strukturen ('structs'), die sowohl die physikalischen Eigenschaften des Labyrinths als auch die Navigationspunkte innerhalb desselben repräsentieren.

Point-Struktur

Die ‘Point’-Struktur repräsentiert einen Punkt innerhalb des Labyrinths und enthält Koordinaten sowie Verknüpfungen zu vorherigen und nächsten Punkten, falls verwendet.

```

1 typedef struct {
2     int x; /* x-Koordinate */
3     int y; /* y-Koordinate */
4     struct Point *prev; /* Zeiger auf vorherigen Punkt im Pfad */
5     struct Point *next; /* Zeiger auf naechsten Punkt im Pfad */
6 } Point;

```

Maze-Struktur

Die ‘Maze’-Struktur beschreibt das Labyrinth selbst. Sie enthält Informationen über die Dimensionen des Labyrinths (Anzahl der Reihen und Spalten), Start- und Endpunkte sowie das eigentliche Gitter, das die Struktur des Labyrinths repräsentiert.

```

1 typedef struct {
2     uint8_t rows; /* Anzahl der Reihen im Labyrinth */
3     uint8_t cols; /* Anzahl der Spalten im Labyrinth */
4     Point start; /* Startpunkt */
5     Point exit; /* Zielpunkt */
6     uint8_t** grid; /* 2D-Array, das das Labyrinth darstellt */
7 } Maze;

```

Queue-Struktur

Für die Speicherung und Verwaltung der zu durchsuchenden Punkte während der Labyrinthlösung wird eine ‘Queue’-Struktur verwendet, die eine Warteschlange implementiert. Sie ist insbesondere für die Implementierung der Breitensuche (BFS) relevant.

```

1 typedef struct {
2     Point* head; /* Zeiger auf den ersten Punkt in der Warteschlange */
3     Point* tail; /* Zeiger auf den letzten Punkt in der Warteschlange */
4     uint16_t size; /* Anzahl der Punkte in der Warteschlange */
5 } Queue;

```

4.4.2 Der Generierungsprozess

Die Labyrinthgenerierung beginnt mit der Initialisierung des Maze-Structs, indem die `initMaze` Funktion aufgerufen wird. Diese Funktion legt die Größe des Labyrinths fest und initialisiert das Gitter mit Wänden (`WALL`).

```

1 void initMaze(Maze* maze, uint8_t rows, uint8_t cols,
2                 uint8_t startX, uint8_t startY,
3                 uint8_t exitX, uint8_t exitY) {
4     maze->rows = rows;
5     maze->cols = cols;
6     // Weitere Initialisierungen...
7 }

```

Nach der Initialisierung beginnt der eigentliche Generierungsprozess, der eine Reihe von Schritten durchläuft, um Wege (PATH) im Labyrinth zu schaffen. Die Funktion `carveMaze` wird für jede Zelle des Labyrinths aufgerufen, beginnend bei der Startposition. Die Funktion wählt zufällig eine Richtung aus und versucht, in dieser Richtung zwei Zellen weit zu "graben" (Wege zu schaffen), vorausgesetzt, die Zielzelle und die dazwischenliegende Zelle sind beide Wände.

```

1 void carveMaze(Maze* maze, uint8_t x, uint8_t y) {
2     int dir = getRand(&rng) % 4; /* Zufällige Richtung */
3     // Versuche, in der gewählten Richtung zu graben
4 }

```

Die zufällige Auswahl der Richtung und das Überprüfen, ob ein Graben möglich ist, gewährleisten, dass das generierte Labyrinth sowohl zufällig als auch lösbar ist.

4.4.3 Schlüsselkonzepte

Ein Schlüsselkonzept in der Labyrinthgenerierung ist die rekursive Backtracking-Methode, die sicherstellt, dass das gesamte Gitter erkundet und ein lösbares Labyrinth erzeugt wird. Jeder Schritt der Generierung wählt zufällig eine neue Richtung, um die Vielfalt und Komplexität des Labyrinths zu erhöhen.

Das Ergebnis dieses Prozesses ist ein Labyrinth mit einem klaren Pfad vom Start zum Ziel, bereit für die Paketdrohne, um das Paket zu liefern. Die Visualisierung dieses Pfades und die Animation der Drohnenbewegung werden durch die nachfolgenden Schritte der Software unterstützt.

4.5 Lösung des Labyrinths

Nachdem das Labyrinth generiert wurde, besteht die nächste Herausforderung darin, einen Weg vom Startpunkt zum Ziel zu finden. Diese Aufgabe wird von der Funktion `solveMaze` in der Datei `maze.c` übernommen. Der Kern dieses Prozesses basiert auf dem Algorithmus der Breitensuche (Breadth-First Search, BFS), einem klassischen Suchalgorithmus, der für die Lösung von Labyrinthen besonders gut geeignet ist.

4.5.1 Der Lösungsalgorithmus

Die Breitensuche beginnt am Startpunkt des Labyrinths und erkundet systematisch alle benachbarten Zellen, bis der Zielpunkt gefunden wird. Dies geschieht durch die schrittweise Exploration aller von einer Zelle aus erreichbaren Nachbarzellen, wobei jede neu entdeckte Zelle in eine Warteschlange eingefügt wird. Die Zellen in der Warteschlange repräsentieren den aktuellen "Rand" der Erkundung.

```

1 void solveMaze(Maze* maze, Queue* path) {
2     // Initialisierung der Warteschlange und Markierung des Startpunkts
3
4     while /* queue nicht leer */ {
5         // Entferne das vorderste Element der Warteschlange
6         // Überprüfe, ob das Ziel erreicht wurde
7         // Füge benachbarte, noch nicht besuchte Zellen der Warteschlange
8         hinzu
9     }
}
```

Jede Zelle wird dabei nur einmal besucht, um sicherzustellen, dass der Algorithmus den kürzesten Pfad zum Ziel findet.

4.5.2 Schlüsselkonzepte

Ein zentrales Konzept der Breitensuche ist die Verwendung einer Warteschlange zur Speicherung der zu erkundenden Zellen. Dies gewährleistet, dass die Zellen in der Reihenfolge ihrer Entdeckung bearbeitet werden, was für die Findung des kürzesten Pfads entscheidend ist. Die Implementierung der Warteschlange erfolgt über die in `queue.c` definierten Funktionen, die eine effiziente Handhabung der Ein- und Ausgabeoperationen ermöglichen.

Ein weiteres wichtiges Element ist die Markierung der besuchten Zellen, um sicherzustellen, dass der Algorithmus nicht in eine Endlosschleife gerät. Diese Markierungen helfen auch dabei, den zurückgelegten Pfad nachzuvollziehen, sobald das Ziel erreicht ist.

4.5.3 Implementierungsdetails

Die Implementierung der Breitensuche in `solveMaze` nutzt zusätzlich zu den Warteschlangenoperationen eine Reihe von Hilfsfunktionen, um benachbarte Zellen zu identifizieren und zu überprüfen, ob sie Teil des Lösungspfads sein können. Diese Überprüfungen beinhalten, ob eine Zelle bereits besucht wurde und ob sie eine Wand oder ein Pfad ist.

```

1 // Beispiel: Hinzufügen einer benachbarten Zelle zur Warteschlange
2 if (maze->grid[adjacentY][adjacentX] == PATH && !visited[adjacentY][
3     adjacentX]) {
4     enqueue(queue, adjacentX, adjacentY);
5     // Markiere die Zelle als besucht
6     visited[adjacentY][adjacentX] = true;
7 }
```

4.5.4 Animation der Drohne auf der LED-Matrix

Nachdem ein Pfad durch das Labyrinth gefunden wurde, wird die Bewegung der Paketdrohne auf der LED-Matrix animiert, um die Lieferung des Pakets visuell darzustellen. Die "Drohne" wird durch einen Pixel in der vorgegebenen Paketfarbe repräsentiert, welcher den gelösten Pfad entlang navigiert.

Darstellung der Drohne

Die Drohne wird als einzelner leuchtender Pixel auf der LED-Matrix dargestellt. Die Farbe dieses Pixels entspricht der Farbe des Pakets, was die Drohne visuell hervorhebt. Während die Drohne den Pfad entlangfliegt, wird ihre aktuelle Position auf der Matrix stetig aktualisiert, um die Bewegung

entlang des Pfades zu simulieren.

Visualisierung des Pfades

Um den Pfad, den die Drohne zurücklegt, besser nachvollziehen zu können, werden bereits besuchte Zellen in einer dunkleren Version der Paketfarbe eingefärbt. Dies schafft einen visuellen Kontrast zwischen dem aktuellen Standort der Drohne und dem bereits zurückgelegten Weg.

Implementierung der Animation

Die Animation wird durch eine Schleife implementiert, welche den gefundenen Pfad Punkt für Punkt durchläuft und die LED-Farben entsprechend anpasst. Die Arrays `color` und `darkColor` speichern die Farben bzw. dunklere Versionen gemäß der Spezifikation der Paketfarben laut Aufgabenstellung.

```
1 for (i = path.size - 1; i > -1; i--) {  
2     ws2812_SPI_pixel(path.p[i].x, path.p[i].y, &color[packageId]);  
3     HAL_Delay(ANIMATION_FRAMETIME_MS);  
4     ws2812_SPI_pixel(path.p[i].x, path.p[i].y, &darkColor[packageId]);  
5 }
```

5 Tests

5.1 Testaufbau

Das Projekt wurde mit drei Nucleo-Boards als Nachbarn, welche alle mit unserem Code geflasht wurden, getestet. Die Baseboards wurden entsprechend in einer Kette zusammengesteckt. Für die Paketweiterleitung waren jeweils Verbindungen vom Labyrinth zu drei Test-Nachbarn und zurück geschaltet.

Nach Beseitigung einiger Fehler wurde das Labyrinth einwandfrei mit den entsprechenden Ein- und Ausgängen generiert, gelöst und die Animation richtig abgespielt.

5.2 Team 13 (Lego)

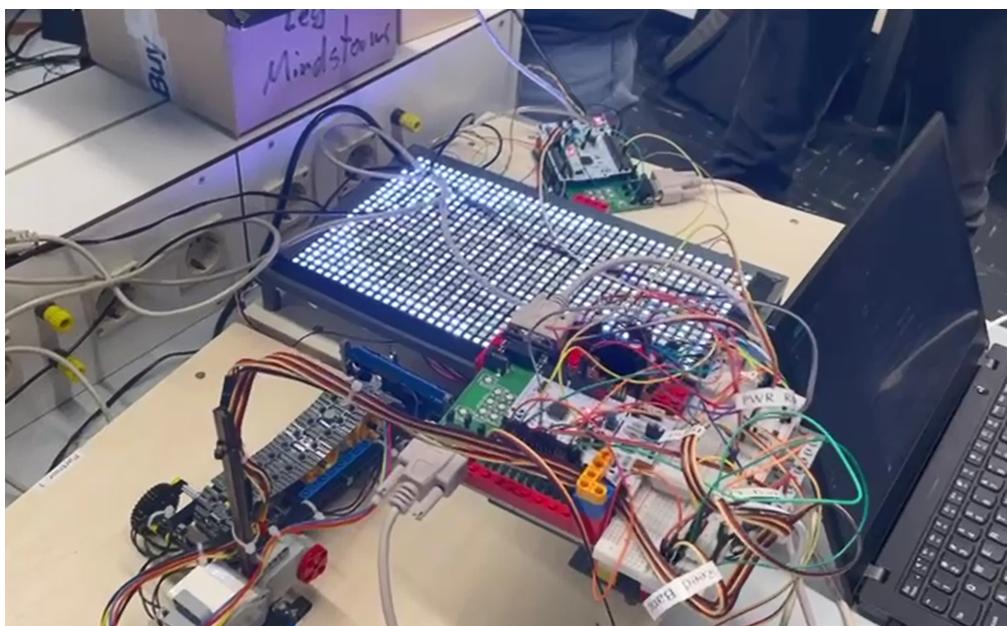


Abbildung 7: Test mit Team 13

Zunächst wurde die Interaktion mit dem System der Team 13 (Lego) getestet (Abb. 7). Dazu wurden die Baseboards entsprechend zusammengesteckt. Der „Master“ wurde dabei an das Baseboard unserer Teams angeschlossen, entsprechend wurde der „Chain End“ Jumper beim Baseboard von Team 13 gesteckt. Im Testprogramm wurden zwei Routen konfiguriert:

```
self.route[0] = [3,13]
self.route[1] = [13,3]
```

Beide Routen funktionierten auf Anhieb einwandfrei:

- das System fuhr zum korrekten Übergabepunkt
- das System wartete, bis das Paket vom Partner übergeben wurde
- das Paket wurde korrekt eingelagert - das System fuhr wieder zum richtigen Übergabepunkt und übergab das Paket erfolgreich an den Nachbarn
- der Nachbar nahm das Paket im richtigen Augenblick entgegen und lagerte es ein.

Obwohl die Paketweiterleitung funktionierte kam es beim Master-Programm zeitweise zur Meldung
ERROR ###.

5.3 Team 13 (Lego) und Team 12 (Roboterarm2)

Anschließend wurden alle das System mit den Aufbauten von Team 13 (Lego) und Team 12 (Roboterarm2) zusammengeschlossen. Folgende Routen wurden konfiguriert:

```
self.route[0] = [3,13,12]
self.route[1] = [12,13,3]
```

Hier funktionierte nur die erste Route. Ein Video dieses Tests ist hier hochgeladen <https://videos.portuus.de/w/9pmaQNVXYw2zNvjc7aKkho> Beim Empfang von Paketen von Team 12 kam es immer wieder zu Problemen, die nicht eindeutig zugeordnet werden konnten. Ebenfalls kam es in den anschließenden Testversuchen immer wieder zu Problemen bei der Übergabe an Team 12, entweder nahm der Roboterarm das Paket zu früh oder das Lego-Paketband nahm das Paket zu früh oder gar nicht entgegen. Mit Team 13 funktionierte auch in dieser Konstellation der Ablauf stehts einwandfrei. Auch hier kam es beim Master-Programm zeitweise zur Meldung **### ERROR ###** und es wurde mit zunehmender Paketanzahl instabiler.

Abbildungsverzeichnis

1	Kontextdiagramm	1
2	Datenflussdiagramm	2
3	Zustandsübergangsdiagramm	2
4	Verdrahtungsplan	6
5	Konstruktionsskizze	7
6	Timings für die WS2812 LEDs [ws2812_SPI]	8
7	Test mit Team 13	15

Tabellenverzeichnis

1	Prozessaktivierungstabelle	5
---	--------------------------------------	---