Заглавна страница (по образец)

Задание (при принтиране се заменя с тази празна страница)

+

Декларация, че съм си написал всичко сам с подпис (имам я в мейла)

Съдържание

1. Увод

Дипломната работа, озаглавена “Система за управление на LED матрица с Bluetooth свързаност,” се фокусира върху решаването на значим технически проблем – безжичното управление на LED матрица, използвана за различни приложения, като визуализиране на информация или декоративно осветление. Този проект има за цел да разработи интегрирана система, която осигурява надеждно и ефективно прехвърляне на данни между потребителското устройство и хардуерната система чрез използването на Bluetooth технология. Приложимостта на проекта обхваща редица области – от домашна автоматизация и управление на рекламни дисплеи до образователни и индустриални приложения.

В хода на разработката са дефинирани три основни задачи, които играят ключова роля за успешното реализиране на проекта. Първата задача включва създаването и внедряването на специализиран алгоритъм, който да осигурява достоверно и устойчиво предаване на данни в безжичен режим. Този алгоритъм трябва да гарантира, че дори при наличие на смущения или загуба на сигнал, данните ще бъдат предадени коректно и навременно. Втората задача се отнася до проектирането на интуитивен и функционален потребителски интерфейс, който да улеснява управлението на LED матрицата. Целта е интерфейсът да бъде достатъчно лесен за използване, дори за потребители без технически опит, като същевременно предоставя достъп до всички необходими функции за контрол и настройка на устройството. Третата задача включва програмирането и интеграцията на хардуера, което да гарантира точното изпълнение на командите и визуализирането на промените върху LED матрицата. Хардуерната част на системата е отговорна за получаването на безжичните команди и тяхната коректна обработка.

Методологията на проекта включва внимателен подбор на технологии, които да отговорят на нуждата от стабилност, ефективност и гъвкавост на разработваната система. Изборът на платформата Arduino Uno R3 се основава на нейната популярност и доказана функционалност в хардуерни проекти от различен мащаб. Платформата предлага стабилност и лесна интеграция с периферни устройства, като същевременно осигурява балансирано съотношение между цена и производителност, което я прави особено подходяща за

проекти с ограничени ресурси. Arduino Uno R3 също така разполага с богата екосистема от библиотеки и поддръжка, което ускорява процеса на разработка и прави системата по-гъвкава за бъдещи надстройки.

За създаването на потребителския интерфейс е използван езикът Kotlin за разработката на Android приложение. Kotlin е предпочетен поради своя модернизиран синтаксис, съвместимост с Java и възможността за бързо и лесно разработване на мобилни приложения. Това позволява на потребителя лесен и интуитивен достъп до функциите на системата, като същевременно осигурява надеждност в комуникацията с LED матрицата. Използването на Bluetooth технологията гарантира стабилна и ефективна комуникация на къси разстояния, което е от съществено значение за безжичното управление на хардуерното устройство. Bluetooth модулът, интегриран в системата, осигурява бърз обмен на данни между мобилното приложение и микроконтролера, като минимизира забавянията и възможността за грешки.

Комбинация между тези технологии – Arduino Uno R3, Kotlin за Android и Bluetooth – осигурява на системата не само надеждност, но и възможност за по-нататъшни подобрения и персонализации според нуждите на потребителя. Тази гъвкавост прави системата подходяща за различни сценарии на употреба, включително индустриални приложения и персонални проекти за управление на LED дисплеи.

2. Литературно проучване

Теоретичната основа на този проект се основава на редица съвременни хардуерни и софтуерни технологии, всяка от които играе ключова роля за успешната реализация на системата за комуникация между смартфон приложение и микроконтролер Atmega 328P, чрез Bluetooth. В този раздел ще разгледаме подробно основните компоненти и технологии, използвани за реализиране на проекта: микроконтролер Atmega 328P, платка Arduino Uno R3, операционна система Android, езика за програмиране Kotlin, както и безжична комуникация чрез Bluetooth.

Atmega328P

Atmega328P е 8-битов микроконтролер, произведен от Atmel, широко използван в различни електронни и роботизирани проекти поради своята ефективност и гъвкавост [1]. Той е сърцето на много Arduino платки, включително Arduino Uno. Микроконтролерът разполага с 32KB флаш памет, 2KB SRAM и 1KB EEPROM, което го прави подходящ за съхранение и изпълнение на прости до средно сложни програми [2]. Освен това, той поддържа до 20 MHz честота на работа, което осигурява стабилна производителност при работа с множество сензори и периферни устройства. Интегрираната архитектура RISC позволява на микроконтролера да изпълнява операции с минимални тактови цикли, което подобрява енергийната ефективност [3]. Atmega328P има богат набор от входно-изходни (I/O) портове, включително 23 програмируеми I/O пина, аналогови входове и PWM изходи. Той също така разполага с комуникационни интерфейси като UART, SPI и I2C, което улеснява интеграцията с други компоненти и модули, като Bluetooth, използван в проекта.

Arduino Uno R3

Arduino Uno R3 е една от най-популярните платки в средата на електрониката и микроконтролерите [4]. Тя е базирана на Atmega328P микроконтролера и предоставя платформа за лесно прототипиране и

разработка на електронни системи. Arduino Uno R3 се отличава със своята простота и лесен достъп до хардуерни и софтуерни ресурси, което я прави подходяща както за начинаещи, така и за опитни разработчици . Платката разполага с 14 цифрови I/O пина, 6 от които поддържат PWM изходи, както и 6 аналогови входа. Освен това, Uno R3 има USB порт за програмиране и комуникация с компютър, както и ICSP хедър за програмиране директно на микроконтролера. Захранването на платката може да бъде осигурено както чрез USB, така и чрез външен източник на енергия (7-12V), което я прави гъвкава в различни условия на работа [5]. Една от ключовите характеристики на Arduino Uno R3 е нейната софтуерна поддръжка. Arduino IDE е лесна за употреба среда за разработка, която използва опростен език, базиран на C++, и предоставя голяма библиотека от предварително написани функции, което значително улеснява работата с периферни устройства и комуникационни модули, като Bluetooth [6].

Android

Android е водеща операционна система за мобилни устройства, разработвана от Google, която се използва в милиарди устройства по целия свят [7]. Тя е базирана на ядрото на Linux и предоставя богата екосистема от софтуерни инструменти и библиотеки, които улесняват разработката на мобилни приложения за широка гама от устройства. Основната архитектура на Android се състои от четири основни слоя: ядрото на Linux, библиотеки и Android Runtime (ART), рамка за приложения и приложения [8]. Тази структура осигурява висока степен на модулност и гъвкавост, което позволява лесно създаване и управление на различни видове мобилни приложения, включително такива, които комуникират с външни устройства чрез Bluetooth [9]. Android SDK (Software Development Kit) предлага широка поддръжка за различни хардуерни функции, включително Bluetooth API, което улеснява интеграцията с външни сензори и микроконтролери, като тези използвани в проекта [10].

Kotlin

Kotlin е модерен език за програмиране, създаден от JetBrains, който е официално поддържан за разработка на Android приложения от Google [11]. Той е предпочитан в много проекти, защото комбинира простотата на синтаксиса с мощни функции като null safety, ламбди и разширени функции, които улесняват разработката и намаляват вероятността за грешки [12]. Едно от основните предимства на Kotlin при разработката на Android приложения е подобрената му поддръжка за асинхронно програмиране чрез корутини [13]. Това е особено важно при работа с външни устройства, като Arduino, където комуникацията по Bluetooth може да доведе до закъснения и трябва да бъде управлявана по оптимален начин, за да не блокира основния потребителски интерфейс.

Bluetooth

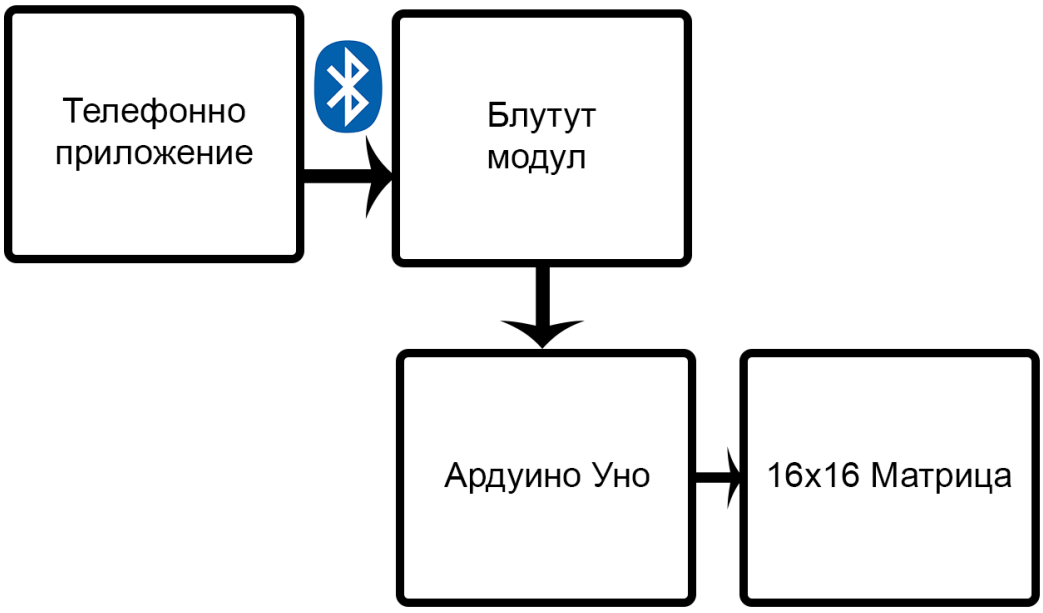
Bluetooth е стандарт за безжична комуникация на къси разстояния, който е широко използван в устройства като мобилни телефони, компютри, слушалки и различни IoT устройства. В контекста на този проект, Bluetooth осигурява връзката между смартфон приложението и Arduino платката, позволявайки предаването на данни и команди в реално време. Технологията Bluetooth работи на честотен диапазон от 2.4 GHz и поддържа скорости на предаване до 3 Mbit/s в стандартния си клас. Основното предимство на Bluetooth в сравнение с други безжични технологии, като Wi-Fi, е неговата ниска консумация на енергия и лесната му интеграция в малки устройства [14].

HC-06

HC-06 е Bluetooth модул, който работи по протокола Bluetooth 2.0 и осигурява безжична комуникация на къси разстояния до 100 метра. Използва се предимно в хоби проекти и инженерни приложения за комуникация с микроконтролери като Arduino чрез UART интерфейс. Модулът работи в диапазона 2.402–2.480 GHz, поддържа скорост до 2.1 Mbps и изисква минимална консумация на енергия, което го прави подходящ за батерийни устройства [15].

3. Проектиране на блокова схема на хардуера

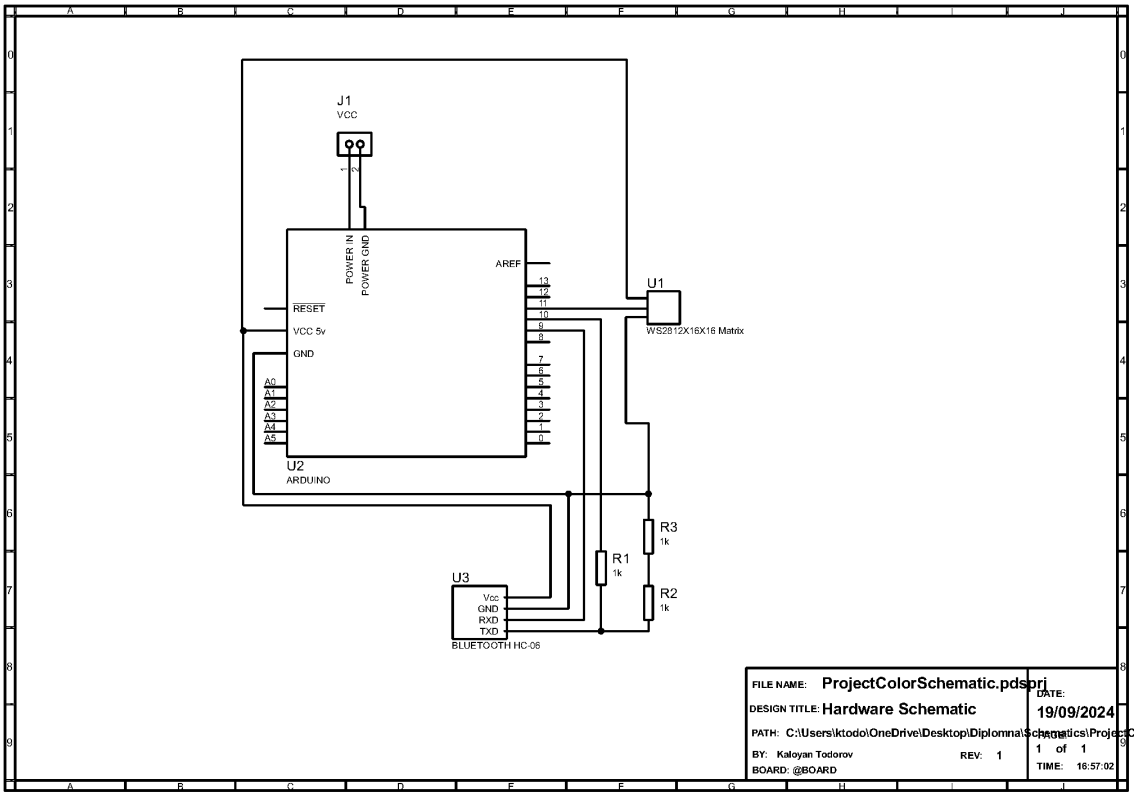
За изпълнение на задачата е нужно да се проектира блокова схема на хардуер. Тя се състои от блок “телефонно приложение”, блок “ардуино”, блок ”блутут модул” и блок “матрица”. Телефонното приложение комуникира с блутут модула на ардуиното, като предадената информация се обработва и светодиодите на матрицата се осветяват в зададените цветове.



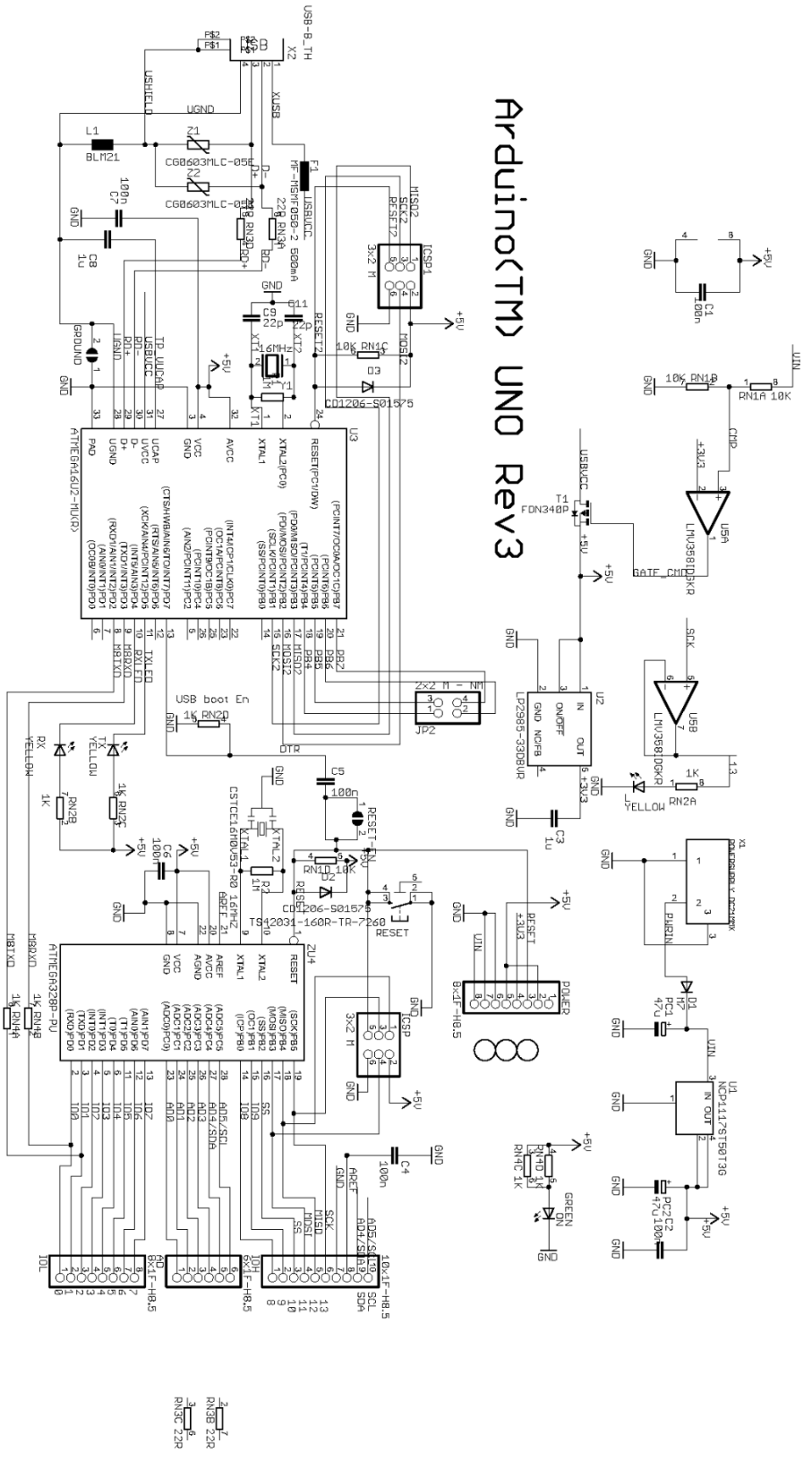
Фиг. 3.1 Блокова схема на хардуера.

4. Проектиране на принципна схема (todo: преработка)

Принципната схема (фиг. 4.1) е проектирана с помощта на софтуерен продукт Proteus 8, създавайки проект в A4 размер с части, репрезентиращи нужния хардуер. В принципната схема са обозначени всичките връзки между пиновете на всички устройства. Тя представлява J1- външно захранване, U1 - Ардуино платка, U2 - 16x16 светодиодна матрица, U3 - Блутут модул HC-06, резисторите R1, R2, R3. На фиг. 4.2 е изобразена принципната схема на Arduino Uno R3 [16].



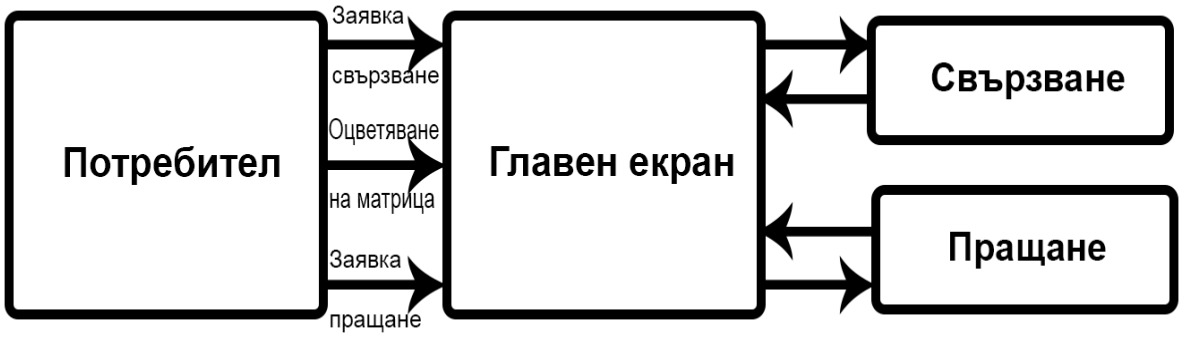
Фиг. 4.1 Принципна схема на хардуера.



Фиг. 4.2 Принципна схема на Arduino Uno R3.

5. Проектиране на блоков алгоритъм на системния и приложния софтуер

5.1. Блоков алгоритъм за приложния софтуер



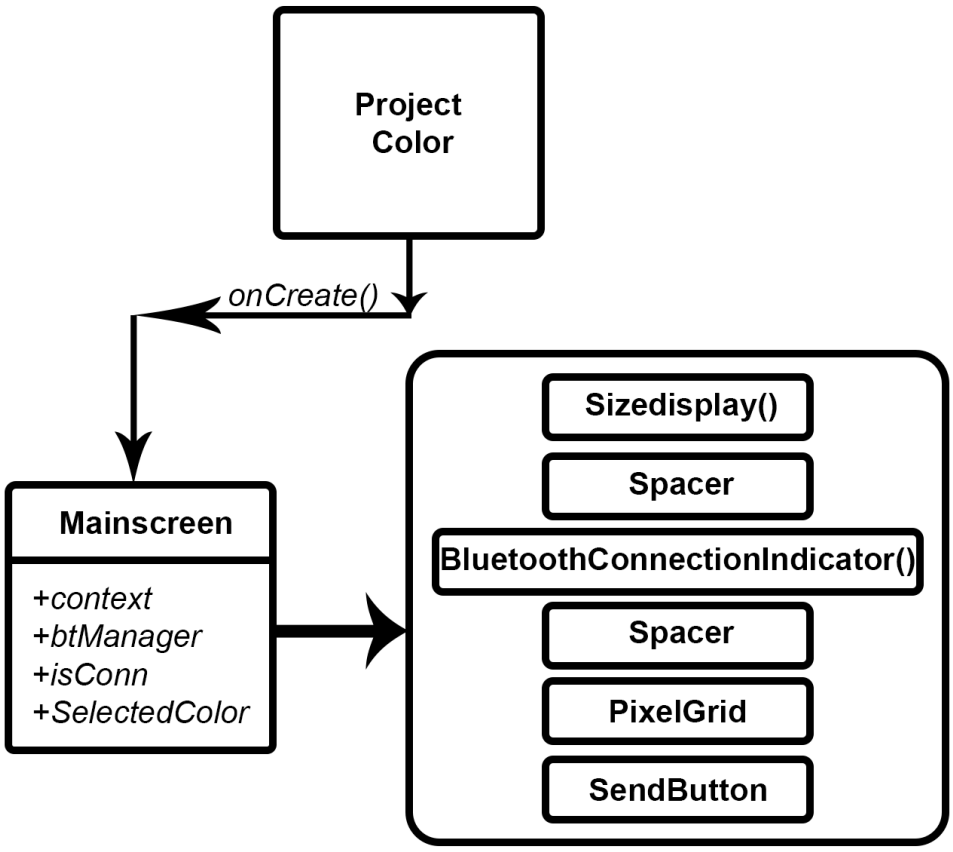
Фиг. 5.1 Блоков алгоритъм на приложния софтуер.

Проектираният блоков алгоритъм за приложния софтуер се състои от няколко основни блокове, всеки от които изпълнява специфична функция в общия процес. Първият блок, “Главен екран”, отговаря за графичното представяне на приложението и взаимодействието с потребителя. Също така разполага с механизъм за динамична промяна на цветовата схема на визуализираната графична матрица. Блокът “Потребител” представлява потребителя на приложението, на който взаимодействията му с блок “Главен екран” се интерпретират като заявки.

Блок “Свързване” осигурява логиката за установяване на Bluetooth връзка, активирана при съответната заявка от блок “Потребител”. Този блок е отговорен за осъществяване на комуникация с избрано външно устройство.

Блокът “Пращане” управлява процеса на предаване на матрична информация към свързаното устройство в отговор на съответната потребителска заявка.

- **Блок “Главен екран”:**

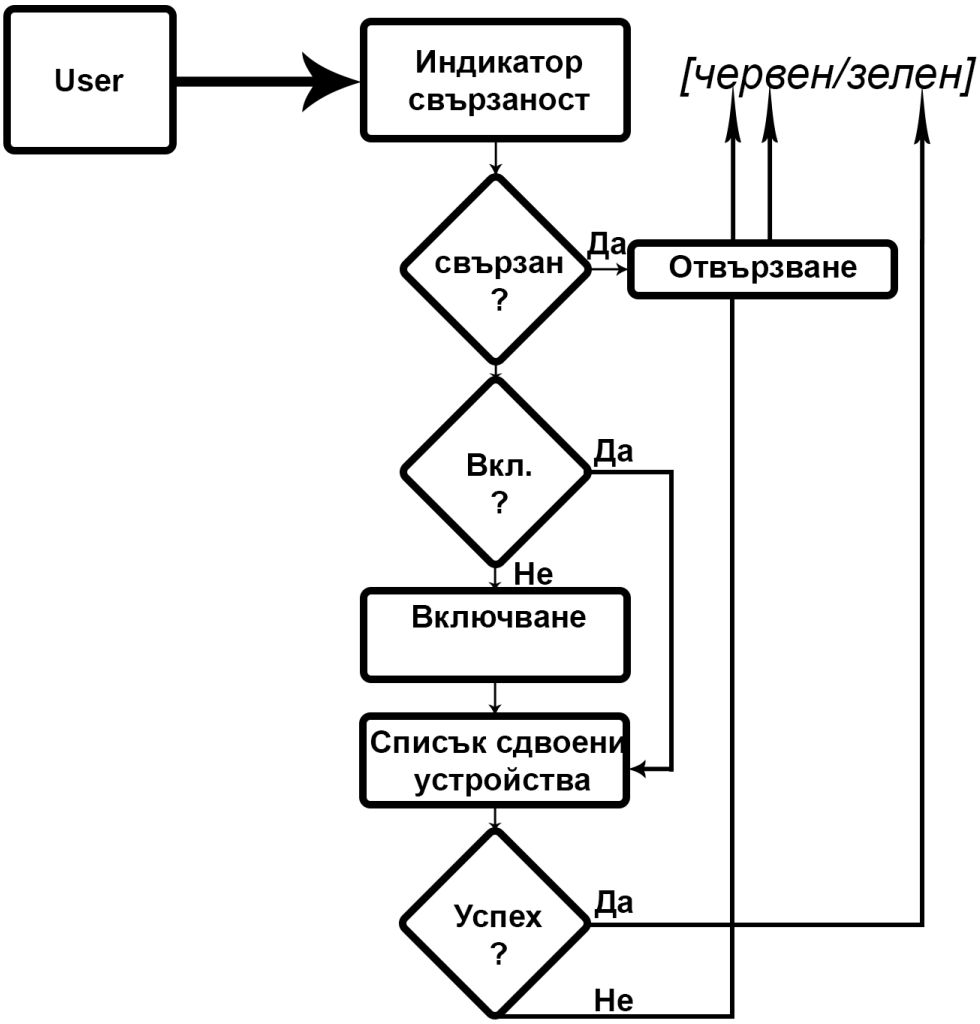


Фиг. 5.2 Блок “Главен екран”.

Блокът "Главен екран" представлява графичното оформление на основния екран на приложението. Той организира и визуализира следните елементи в колона: информация за размерите на матрицата, разстояние между елементите, бутон-индикатор за Bluetooth свързаност, разстояние, ред от бутони за избор на цвят, пикселна решетка, която представлява матрицата и е съставена от бутони, както и ред с бутони за заявка за изпращане на картинката от решетката - матрицата.

Основната функционалност на матрицата е свързана с възможността при натискане на произволен бутон, той да се оцвети в предварително избрания цвят от реда за избор на цвят.

- **Блок “Свързване”:**



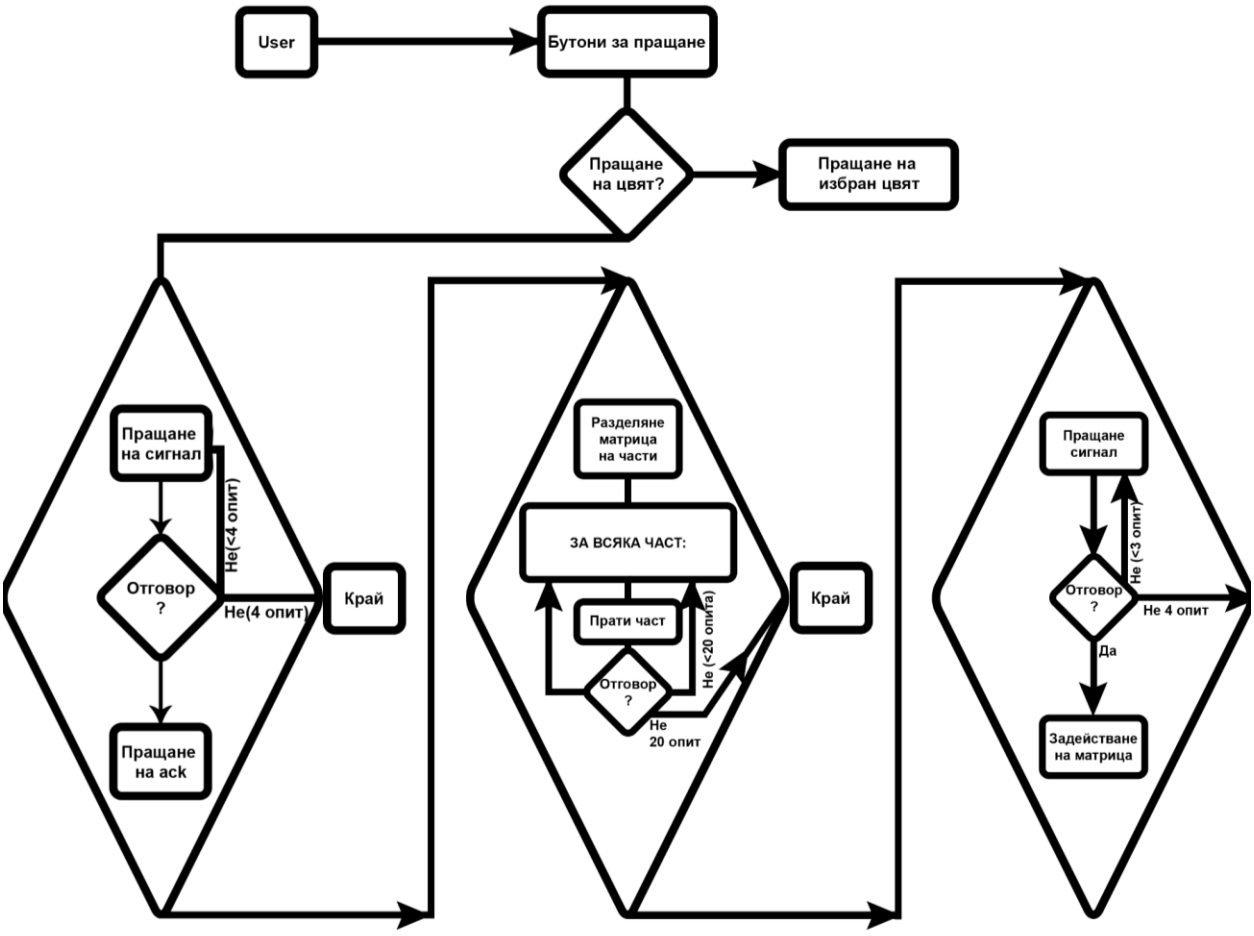
Фиг. 5.3 Алгоритъм на Блоков “Свързване”.

Заявката за свързване, инициирана от потребителя, се активира чрез натискане на бутона "Индикатор за свързаност". Този бутон има две състояния: червено (когато няма установена връзка с устройство) и зелено (когато връзката с Bluetooth устройство е успешно установена). В зависимост от текущото му състояние се изпълняват следните действия:

Ако вече е установена връзка (индикаторът е в зелено), натискането на бутона прекъсва връзката, като индикаторът се връща в червено;

Ако няма установена връзка (индикаторът е в червено), приложението първо проверява дали Bluetooth модулът на устройството е активен. Ако не е, на потребителя се показва диалогов прозорец с опция за включване на Bluetooth. При активиран Bluetooth (или ако потребителят избере да го включи), на екрана се появява списък със сдвоените устройства. След избор на устройство, приложението прави опит за установяване на връзка. При успешно свързване, индикаторът става зелен, а при неуспех остава червен.

- **Блок “Пращане”:**



Фиг. 5.4 Алгоритъм на блок “Пращане”.

За използването на блок "Пращане" е необходимо предварително установена връзка с Bluetooth устройство. При наличие на такава връзка, могат да бъдат изпълнени две основни операции: изпращане на матрична картинка или изпращане на избран цвят. Заявката за изпращане на матрична картинка се активира чрез натискане на бутона "Send", докато заявката за изпращане на избран цвят се реализира чрез натискане на съответния бутон за цвят.

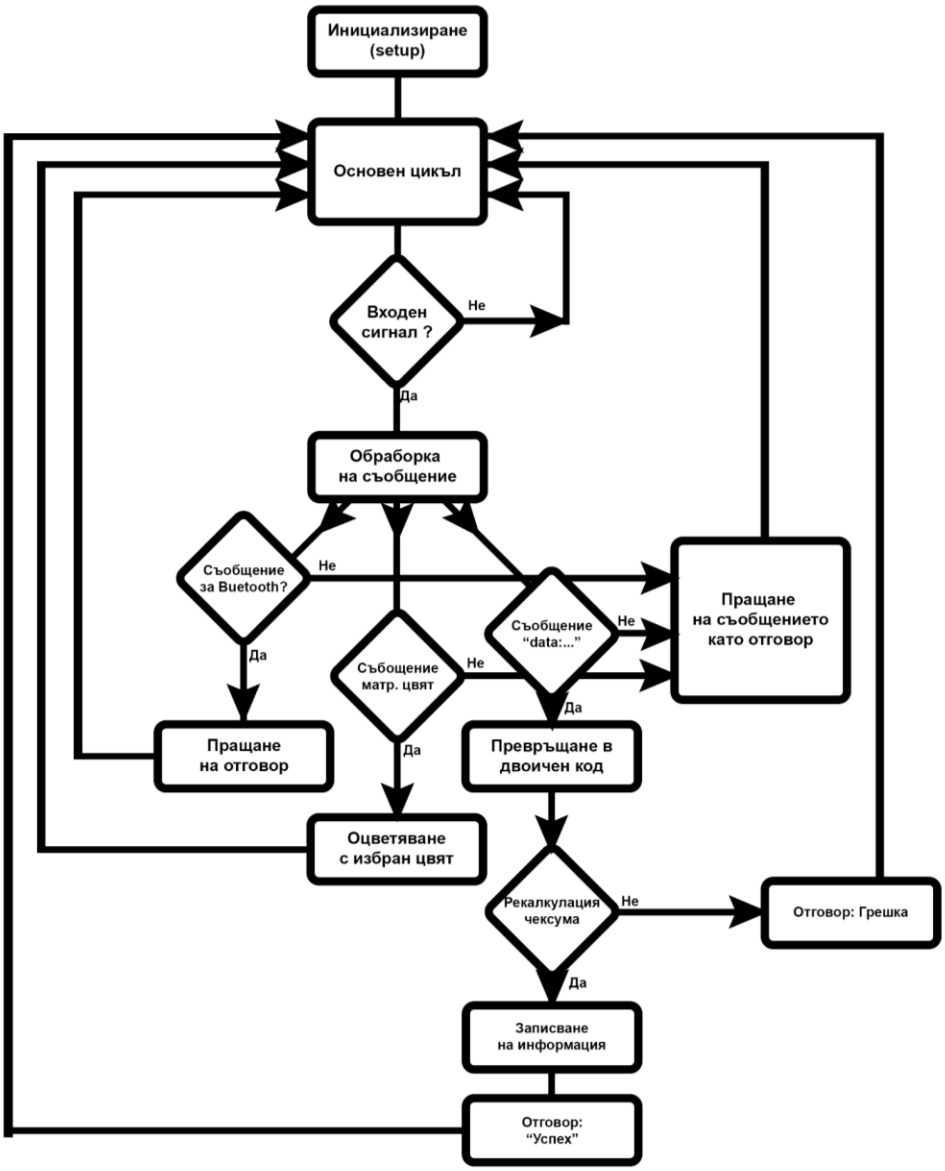
При изпращане на конкретен цвят, избраният цвят се предава към свързаното устройство чрез Bluetooth комуникация.

За изпращането на матричната картинка, първо се активира блокът "Ръкостискане", който оперира подобно на тристепенния механизъм за ръкостискане на мрежовия протокол TCP за надеждна комуникация. Първо се изпраща сигнал за синхронизация към устройството, като се очаква отговор. При неполучен отговор се правят до два допълнителни опита. Ако след всички опити отговорът липсва, алгоритъмът се прекратява. При получаване на правилен отговор се изпраща потвърждение и алгоритъмът преминава към блок "Пращане на Матрица".

В блок "Пращане на Матрица" матрицата се обработва ред по ред. Всеки ред се разделя на части, като всяка част се изпраща последователно, след което се очаква потвърждение от устройството. При отрицателен или липсващ отговор се правят до 19 опита за изпращане на конкретната част. След успешно изпращане на всички части от матрицата, алгоритъмът навлиза в блок "Терминация".

Блок "Терминация" изпраща сигнал за финализиране на комуникацията, който също така служи за "активиране" на матрицата на свързаното устройство. При липса на потвърждение след първоначалния сигнал се правят до два допълнителни опита. Ако и след това няма отговор, се показва индикация за неуспех на потребителя. При успешно изпратен сигнал и потвърден отговор, операцията приключва, но връзката с устройството остава активна.

5.2. Блоков алгоритъм за системния софтуер



Фиг. 5.5 Блоков алгоритъм за системния софтуер".

Блоковият алгоритъм на системния софтуер започва с блок "Инициализиране", в който се инициализират необходимите компоненти, включително Bluetooth комуникацията и матрицата от LED светлини. След тази инициализация, алгоритъмът преминава в основен цикъл, който непрекъснато следи и събира входящите данни

от Bluetooth комуникацията във вид на символен низ, ако такива данни са налични. При получаване на символ '\0' (край на символен низ) или '\n' (нов ред), събраното съобщение се предава за обработка в блока "Обработка на съобщението".

В зависимост от съдържанието на съобщението се изпълняват следните действия:

- Ако съобщението е част от Bluetooth комуникационни сигнали, изпратени от приложението, софтуерът изпраща необходимия отговор.
- Ако съобщението съдържа команда за оцветяване на матрицата, софтуерът изпълнява тази команда.
- Ако съобщението започва с "data:", то се предава за обработка в блок "Обработка на част".
- Ако съобщението не отговаря на нито един от тези критерии, то се изпраща обратно към източника.

● Блок "Обработка на част":

Блокът "Обработка на част" има за цел да обработи частта от данните, изпратена от приложния софтуер. Тази част представлява сериализирана информация за няколко последователни пиксела от матрицата, съдържаща също така контролна сума (чексум) в края на съобщението. След като съобщението бъде конвертирано в двоичен формат, чексумът се преизчислява. При коректно преизчисление чексумът се равнява на нули в рамките на зададения размер на блока. Ако проверката на чексумът е успешна, информацията от съобщението се записва в масива на физическата матрица от пиксели, а чрез Bluetooth се изпраща потвърждение за правилно получена част. В случай на несъответствие в чексумата, се изпраща съобщение за грешка, сигнализиращо за некоректно изпратена част.

6. Проектиране на системния и приложния софтуер

6.1. Проектиране на приложния софтуер

За изготвянето на програмното решение на приложния софтуер е избрана интегрираната средата за разработка на приложения Android Studio. Тъй като тя е официалната среда за разработване на Android приложения и се поддържа от Google, този избор осигури пълна съвместимост с всички необходими библиотеки и инструменти.

Изготвяне на приложението става със създаване на проект в **Android Studio** от тип **Empty Activity** с минимално **API 31** (Android 12).

Програмният код на приложението започва с основния клас за работа с интерфейса, а именно **Activity** класа. **Activity** е ключов компонент на всяко **Android** приложение, като начинът, по който се стартират и свързват различните активности, е основен елемент от модела на **Android** платформата. За разлика от други програмни парадигми, при които приложенията се стартират с метод **main()**, **Android** системата изпълнява кода в екземпляр на **Activity**, като извиква специфични методи за обратна връзка (**callback methods**), съответстващи на различни етапи от нейния жизнен цикъл.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            ProjectColorTheme {
                Surface(
                    modifier = Modifier.fillMaxSize()
                ) {
                    MainScreen()
                }
            }
        }
    }
}
```

Оттук следва основният клас **MainActivity**, в който започва реалната работа на приложението. Той наследява класа **ComponentActivity**, който предоставя основни функционалности за управление на жизнения цикъл на активността и взаимодействието с потребителския интерфейс. **MainActivity** служи като основната точка, от която се управлява логиката на приложението и се обработват събитията, свързани с интерфейса и потребителското взаимодействие. Чрез наследяването на **ComponentActivity**, приложението получава достъп до стандартните методи за управление на жизнения цикъл, като **onCreate()**, **onStart()**, **onResume()** и други, които осигуряват плавно управление и координация на различните състояния на активността.

В неговия метод **onCreate()** се инициират основните компоненти на потребителския интерфейс и се задават ключови настройки за визуалното оформление на приложението. Първо, се извиква методът

enableEdgeToEdge(), който активира режим за показване на съдържанието на цял екран, премахвайки стандартните рамки на екрана.

След това, с помощта на **setContent{}** се задава съдържанието на активността, използвайки **Jetpack Compose** – декларативна система за изграждане на UI компоненти в Android. Темата на приложението се дефинира чрез **ProjectColorTheme{}**, която осигурява унифициран визуален стил на потребителския интерфейс. Основната повърхност, представена чрез **Surface**, се модифицира с **Modifier.fillMaxSize()**, за да заеме целия екран.

Накрая, в рамките на тази повърхност, се извиква методът **MainScreen()**, който управлява и визуализира основния интерфейс на приложението, съдържащ основните функционалности и интерактивни елементи.

```
fun MainScreen() { }
```

Функцията **MainScreen()** представлява главния потребителски интерфейс на приложението, което включва Bluetooth свързаност и манипулация на пикселна решетка. Тя управлява цялостното оформление и интегрира няколко ключови компонента, които позволяват на потребителя да взаимодейства с различни функции на приложението.

При изпълнението на **MainScreen()** се извършват няколко важни действия:

- Първоначално се извлича текущият **Context** на приложението, чрез който се инициализира **BluetoothManager**, отговорен за обработката на Bluetooth операциите.
- След това се поддържа състояние за 16x16 RGB пикселна решетка, като се използва **mutableStateOf**, който позволява

динамично проследяване и обновяване на матрицата на пикселите в реално време.
- Проследява се и състоянието на Bluetooth свързаността чрез променлива **isConnected**, която представлява **mutable state** и се обновява спрямо текущата връзка.
- Управлява се също така и състоянието на текущо избрания цвят, който по подразбиране е зададен на **Color.Red**. Този цвят представлява избрания цвят, с който се оцветяват елементите от пикселната решетка.

Потребителският интерфейс на **MainScreen()** е структуриран чрез **Scaffold**, който осигурява цялостното оформление и съдържа следните ключови компоненти:

- **SizeDisplay**, който показва информация, свързана с размера на мрежата и различни параметри, касаещи пикселите.
- **BluetoothConnectionIndicator**, който предоставя възможност на потребителя да се свърже или да прекъсне връзката с Bluetooth устройство.
- **ColorPickerButtons**, който съдържа бутони за избор на цвят, позволявайки на потребителя да актуализира състоянието на избрания цвят.
- **PixelGrid**, който визуализира решетка от пиксели, позволявайки взаимодействие и манипулация на пикселите въз основа на избрания цвят.
- **SendButton**, който при натискане изпраща текущото състояние на пикселната мрежа чрез Bluetooth връзката.

Функцията **MainScreen()** гарантира, че всички потребителски действия – като свързване към Bluetooth, избор на цветове и изпращане на данни – се обработват ефективно, като поддържа коректни състояния в потребителския интерфейс и осигурява плавно взаимодействие с различните компоненти.

```
class BluetoothManager(private val context: Context) { }
```

Класът **BluetoothManager** предоставя пълно управление на Bluetooth операциите в Android приложения, като опростява взаимодействията чрез различни методи за свързване, откриване на устройства и предаване на данни. Ето детайлно описание на функциите, включени в класа:

Конструктор:

- **context**: Контекстът на Android, използван за достъп до системни ресурси.

Полета:

- **bluetoothAdapter**: Адаптер за основна Bluetooth функционалност.
- **_discoveredDevices**: **MutableLiveData**, съхраняваща набор от открити Bluetooth устройства.
- **discoveredDevices**: **LiveData**, която осигурява достъп до откритите устройства.
- **discoveredDevicesSet**: Набор от **BluetoothDevice**, който съхранява откритите устройства.
- **myUUID**: Уникален идентификатор за създаване на RFCOMM Bluetooth сокети.
- **connectJob** и **sendJob**: **Coroutine** задачи за управление на свързването и изпращането на данни.
- **connectionTimeout**: Период на изчакване за Bluetooth връзка.
- **bluetoothSocket**: **BluetoothSocket**, използван за комуникация с устройството.
- **outputStream** и **inputStream**: Потокове за изпращане и получаване на данни.

Методи:

hasBluetoothSupport(): Проверява дали устройството поддържа Bluetooth.

isBluetoothEnabled(): Проверява дали Bluetooth е активиран.

enableBluetooth(launcher: ActivityResultLauncher<Intent>): Стартира интент за активиране на Bluetooth, ако не е активиран.

hasPermissions(): Проверява дали са дадени необходимите Bluetooth разрешения, като отчита версията на Android.

requestPermissions(activity: Activity, requestCode: Int): Изисква необходимите Bluetooth разрешения от потребителя.

getPairedDevices(): Връща набор от сдвоени Bluetooth устройства, като хвърля **SecurityException**, ако липсват разрешения.

startDiscovery(): Стартира процеса на откриване на Bluetooth устройства, като автоматично го спира след 10 секунди.

stopDiscovery(): Спира текущия процес на откриване и анулира регистрацията на приемника за откриване.

pairDevice(device: BluetoothDevice): Стартира процеса на сдвояване с дадено устройство, ако не е вече сдвоено.

connectToDevice(device: BluetoothDevice, onConnectionResult: (Boolean) -> Unit): Опитва да се свърже с определено устройство, като връща резултата чрез callback.

isConnected(): Проверява дали в момента е установена Bluetooth връзка.

sendData(message: String): Изпраща низово съобщение до свързаното Bluetooth устройство.

receiveData(timeoutMillis: Long = 5000L): Чака за получаване на данни от свързаното устройство в рамките на зададено време.

cancelConnection(): Прекратява връзката, затваря Bluetooth сокета и освобождава ресурсите.

BluetoothSocket?.closeSilently(): Безопасно затваря Bluetooth сокета, като логва евентуални грешки.

```
data class PixelData(
    val red: Float,
    val green: Float,
    val blue: Float,
)
```

Класът **PixelData** е **data class**, която представлява информация за цвета на един пиксел във формат RGB.

Включва следните полета:

- **red**: **Float** стойност, която представлява червения компонент на цвета на пиксела, в диапазона от 0.0 до 1.0.
- **green**: **Float** стойност, представляваща зеления компонент на цвета на пиксела, в диапазона от 0.0 до 1.0.
- **blue**: **Float** стойност, представляваща синия компонент на цвета на пиксела, в диапазона от 0.0 до 1.0.

Обектите от този клас се използват в рамките на RGB матрица, за да дефинират цвета на всеки пиксел, осигурявайки прецизен контрол върху дисплея и манипулацията на пикселни данни в приложението.

```
class RGBMatrix (
    val width: Int,
    val height: Int,
) { }
```

Класът **RGBMatrix** представлява двумерна матрица от пиксели, създадена специално за управление и манипулация на RGB цветови

данни. Той съхранява и извлича цветова информация за решетка от пиксели.

Полета:

- **width**: Ширината на пикселната решетка (брой колони).
- **height**: Височината на пикселната решетка (брой редове).
- **data**: Частен масив от обекти **PixelData**, който съхранява RGB цветови данни за всеки пиксел в решетката. Масивът е инициализиран с черни пиксели (стойност 0.0 за компонентите червено, зелено и синьо).

Основни методи:

- **getPixel(x: Int, y: Int): PixelData**: Извлича **PixelData** за пиксела, разположен на зададените координати (x, y). Хвърля **IndexOutOfBoundsException**, ако координатите са извън границите на матрицата.
- **setPixel(x: Int, y: Int, pixel: PixelData)**: Задава цвета на пиксела на зададените координати (x, y) със стойността на дадения **PixelData**. Хвърля **IndexOutOfBoundsException**, ако координатите са извън границите на матрицата.

Частен помощен метод:

- **isValidIndex(x: Int, y: Int): Boolean**: Проверява дали дадените координати (x, y) са в допустимия диапазон на размерите на матрицата. Връща *true*, ако координатите са валидни, в противен случай връща *false*.

```
@Composable
fun SizeDisplay() { }
```

Функцията **SizeDisplay()** е композируем компонент, който показва размерите на пикселната мрежа в потребителския интерфейс. Тя предоставя опростено и центрирано текстово представяне на размерите на решетката, които в случая са "16x16".

При изпълнението на **SizeDisplay()** се извършват следните стъпки:

- Използва се **Row** композируем компонент, за подредбата на своите дъщерни елементи хоризонтално.
- Текстът е центриран хоризонтално в наличното пространство, чрез използване на **Spacer** компоненти с еднаква тежест от двете страни на текста, което осигурява перфектно центриране на текста "16x16" в реда.
- Размерът на решетката се показва чрез **Text** композируем компонент, който е стилизиран с **titleLarge** типографията от **MaterialTheme**.

```
@Composable
fun BluetoothConnectionIndicator(
    modifier: Modifier = Modifier,
    bluetoothManager: BluetoothManager,
    isConnected: Boolean,
    onConnectClick: (BluetoothDevice) -> Unit,
    onDisconnectClick: () -> Unit
) { }
```

Функцията **BluetoothConnectionIndicator()** е композируем компонент, който осигурява потребителски интерфейс за управление на Bluetooth свързаността в приложението. Тя позволява на потребителите да се свързват или прекъсват връзката с Bluetooth устройство, като визуализира текущия статус на връзката.

Функцията включва следните параметри:

- **modifier**: Модификатор, използван за персонализиране на външния вид и оформлението на компонента.
- **bluetoothManager**: Екземпляр на **BluetoothManager**, който обработва Bluetooth операциите.
- **isConnected**: Булева променлива, представляваща текущия статус на връзката (true, ако е свързано, false в противен случай).
- **onConnectClick**: Callback функция, която се извиква при заявка за свързване с Bluetooth устройство.
- **onDisconnectClick**: Callback функция, която се извиква при заявка за прекъсване на връзката с Bluetooth устройство.

Потребителският интерфейс е структуриран в **Box** композируем компонент, който служи като интерактивна зона. Фонът на кутията се променя според състоянието на връзката – зелен за свързано състояние и червен за прекъсната връзка. Вътре в кутията се съдържат следните елементи:

- **Text** елемент, който показва "Connected" или "Disconnected" в зависимост от статуса на връзката.
- Вторичен **Text** елемент, който предоставя насока за потребителя ("Tap to disconnect" или "Tap to connect").
- Опционално съобщение за грешка, ако опитът за свързване се провали.

При натискане на кутията, функцията проверява текущото състояние на Bluetooth:

- Ако устройството е свързано, се активира **onDisconnectClick** callback и се прекратява връзката чрез **bluetoothManager**.
- Ако устройството не е свързано, се проверява дали Bluetooth е поддържан и дали има съответните разрешения. При нужда, Bluetooth се активира, след което започва откриване на устройства и се показва диалог с наличните устройства за свързване.

Когато променливата **showDevicesDialog** е активна, се показва **DevicesDialog**, който позволява на потребителя да избере и да се свърже с Bluetooth устройство. Функцията се грижи за целия процес на свързване, като актуализира потребителския интерфейс и показва съобщения за зареждане или грешки при свързването, ако е необходимо.

```
@Composable
fun DevicesDialog(
    pairedDevices: Set<BluetoothDevice>,
    onDismissRequest: () -> Unit,
    onConnectClick: (BluetoothDevice) -> Unit
) { }
```

Функцията **DevicesDialog()** е композируем компонент, който показва диалог с списък от вече свързани и новооткрити Bluetooth устройства, като предоставя на потребителя възможност да избере устройство за свързване.

Функцията включва следните параметри:

- **pairedDevices**: Множество от обекти **BluetoothDevice**, представляващи устройствата, които вече са сдвоени с хоста.
- **onDismissRequest**: Callback функция, която се извиква, когато диалогът се затвори.
- **onConnectClick**: Callback функция, която се извиква, когато потребителят избере устройство за свързване.

Диалогът е структуриран чрез **Column** композируем, който съдържа:

- Заглавен текст "Paired Devices", който етикетира списъка със сдвоени устройства.

- **LazyColumn**, който визуализира сдвоените устройства, като всяко устройство се представя чрез **DeviceItem** композируем компонент.
- Заглавен текст "Available Devices" (в момента коментиран), който етикетира списъка с открити устройства.
- **LazyColumn** (също коментиран), който би трябвало да визуализира откритите устройства, като всяко устройство се рендерира чрез **DeviceItem**.

```
@Composable
fun DeviceItem(
    device: BluetoothDevice,
    onConnectClick: (BluetoothDevice) -> Unit
) { }
```

Функцията **DeviceItem()** е композируем компонент, който показва информация за едно Bluetooth устройство, заедно с бутон за стартиране на връзка към това устройство.

Функцията включва следните параметри:

- **device**: Обект **BluetoothDevice**, който представлява Bluetooth устройството, което ще се покаже.
- **onConnectClick**: Callback функция, която се извиква, когато потребителят натисне бутона "Connect".

Потребителският интерфейс е структуриран чрез **Row** композируем компонент, който съдържа следните компоненти:

- **Column**, който показва името на устройството (или "Unknown Device", ако името не е налично) и неговия адрес.
- **Button**, който при натискане извиква **onConnectClick** callback с асоциираното устройство.

```
@Composable
fun ColorPickerButtons(
    modifier: Modifier = Modifier,
    onColorSelected: (Color) -> Unit
) { }
```

Функцията **ColorPickerButtons()** представлява композируем компонент, който визуализира хоризонтален ред от бутони за избор на цветове, като позволява на потребителя да избере цвят от предварително дефиниран списък. Избраният цвят се подчертава, а при избор на нов цвят се извиква callback функция.

Параметри:

- **modifier**: **Modifier**, използван за настройка на оформлението и външния вид на реда. По подразбиране е **Modifier**, което не прилага никакви модификации.
- **onColorSelected**: Callback функция, която се активира всеки път, когато потребителят избере цвят. Тя приема избрания **Color** като аргумент.

Състояние:

- **selectedIndex**: Променлива, която проследява индекса на текущо избрания цвят. По подразбиране е зададена на 0 (първият цвят от списъка).
- **colors**: Списък от обекти **Color**, представляващ наличните цветове за избор. Включва цветовете: **Red**, **Green**, **Blue**, **White**, **Black**, и **LightGray**.

Структура на потребителския интерфейс:

- Функцията използва **Row** композируем, за да подреди бутоните хоризонтално, центрирани в рамките на родителя, с **16.dp** отстояние отляво и отдясно, и **8.dp** отгоре и отдолу.

- Вътре в **Row**, се използва **SingleChoiceSegmentedButtonRow**, за да държи индивидуалните **SegmentedButton** елементи, всеки представляващ опция за цвят.
- За всеки цвят от списъка **colors**, се създава **SegmentedButton** с характеристики като:
    - **border**: Граница, която съответства на цвета на бутона, с дебелина от **2.dp**.
    - **onClick**: Действие, което актуализира **selectedIndex** и извиква **onColorSelected** с избрания цвят.
    - **colors**: Определя визуалния вид на бутона, като задава **activeContainerColor** на цвета на бутона, а **inactiveContainerColor** – на прозрачен.
    - **selected**: Булева стойност, която показва дали бутонът е избран, базирано на **selectedIndex**.

Тази функция осигурява интуитивен и лесен за използване интерфейс за избор на цвят, подходящ за приложения за рисуване или персонализиране на потребителския интерфейс.

```
@Composable
fun PixelGrid(
    modifier: Modifier = Modifier,
    size: Int = 16,
    selectedColor: State<Color?>,
    matrix: MutableState<RGBMatrix>
) { }
```

Функцията **PixelGrid()** е композируем компонент, който визуализира интерактивна мрежа от пиксели, като позволява на потребителите да "рисуват" всеки пиксел чрез избор на цвят и клик върху съответната клетка на мрежата. Размерът на мрежата и състоянието на цветовете са адаптивни, а оцветените пиксели се съхраняват в RGB матрица.

Параметри:

- **modifier**: **Modifier**, използван за персонализиране на външния вид и подредбата на мрежата. Стойността по подразбиране е **Modifier**.
- **size**: Цяло число, представляващо размера на мрежата (ширина и височина в брой пиксели). По подразбиране е 16.
- **selectedColor**: **State<Color?>**, представляващ текущо избрания цвят за рисуване. Ако е null, се използва черен цвят.
- **matrix**: **MutableState<RGBMatrix>**, представляващ структурата от данни, която съдържа цветните стойности на пикселите в мрежата.

Структура на потребителския интерфейс:

- Функцията създава **Box** композируем, който определя външната граница на мрежата, добавяйки отстояние, пропорция на аспекта, заоблени ъгли и бяла граница.
- Вътре в **Box**, **Column** се използва за вертикално подреждане на **Row** композируеми, като всяка **Row** представлява ред от пиксели.
- Всеки ред съдържа серия от **PixelButton** композируеми, по един за всеки пиксел в мрежата, подредени хоризонтално.
- Размерът на мрежата е определен от параметъра size, като се генерира мрежа от size x size пиксели. **PixelButton** за всяка клетка се създава на базата на column и row индекси, като цветът на бутона се актуализира динамично при клик, използвайки избрания цвят.

```
@Composable
fun PixelButton(
        modifier: Modifier = Modifier,
        column: Int,
        row: Int,
        selectedColor: Color?,
        matrix: MutableState<RGBMatrix>
) { }
```

Функцията **PixelButton()** е композируем компонент, който представлява отделен бутон в пикселната мрежа, позволявайки на потребителите да "оцветяват" конкретен пиксел чрез избор на цвят и клик върху бутона.

Параметри:

- **modifier**: **Modifier** за прилагане към този бутон. Използва се за персонализиране на външния вид и подредбата на бутона. Стойността по подразбиране е **Modifier**.
- **column**: Цяло число, представляващо колонния индекс на този пиксел в мрежата.
- **row**: Цяло число, представляващо редовия индекс на този пиксел в мрежата.
- **selectedColor**: Nullable **Color**, представляващ текущо избрания цвят за рисуване. Ако е null, се използва черен цвят.
- **matrix**: **MutableState<RGBMatrix>**, която съхранява RGB стойностите на всички пиксели в мрежата и позволява функцията да актуализира цвета на конкретен пиксел.

Функционалност:

- Функцията поддържа **mutable state buttonColor**, която първоначално е зададена на черен цвят (**Color.Black**).
- Когато бутонът се натисне, той актуализира **buttonColor** към текущо избрания цвят или черен, ако няма избран цвят.
- За отстраняване на грешки, избраният цвят се записва в логовете.
- Стойността на цвета се конвертира в обект **PixelData**, съдържащ червени, зелени и сини компоненти на цвета.
- Обектът **PixelData** се съхранява в **matrix** на зададените колона и ред чрез функцията **setPixel**.
- Външният вид на бутона се обновява, за да отрази текущия цвят на бутона, като бутонът се стилизира с правоъгълна форма чрез задаване на радиус на ъглите от 0.dp.

```
@Composable
fun SendButton(
    modifier: Modifier = Modifier,
    matrix: MutableState<RGBMatrix>,
    bluetoothManager: BluetoothManager,
) { }
```

Функцията **SendButton()** е композируем компонент, който визуализира ред с бутони за изпращане на данни от пикселната решетка и команди за специфични цветове чрез Bluetooth. Бутоните са активирани или деактивирани в зависимост от статуса на Bluetooth връзката.

Параметри:

- **modifier**: **Modifier**, приложен към реда с бутоните, използван за персонализиране на външния вид и подредбата на бутоните. Стойността по подразбиране е **Modifier**.
- **matrix**: **MutableState<RGBMatrix>**, представляваща пикселните данни от мрежата. Тези данни се изпращат към Bluetooth устройството при натискане на бутона "Send".
- **bluetoothManager**: Екземпляр на класа **BluetoothManager**, който управлява Bluetooth връзката и предаването на данни.

Структура на потребителския интерфейс:

- Функцията създава **Row** композируем компонент, който съдържа няколко бутона с различна функционалност:
    - **"Send"** бутонът заема повече пространство (**weight** 2) и инициира процеса на изпращане на данни от пикселната мрежа чрез Bluetooth при натискане.
    - Цветовите бутони (**Red**, **Green**, **Blue**, **Black**, **White**), всеки от които изпраща съответната цветова команда към Bluetooth устройството при натискане.

Функционалност на бутоните:

- Активирането на всеки бутон зависи от това дали е установена Bluetooth връзка, което се проверява чрез **bluetoothManager.isConnected()**.
- **"Send"** бутонът стартира **Coroutine**, който изпълнява функцията **handshakeSendPixelQaurterRows**, за да предаде данните от мрежата.
- Цветовите бутони използват функцията **sendColor** за изпращане на специфична цветова команда към Bluetooth устройството.
- Всеки бутон е стилизиран с цветове, зададени чрез **ButtonDefaults.buttonColors**, съответстващи на техния цвят.

```
fun sendColor(color: String, bluetoothManager: BluetoothManager) { }
```

Функцията **sendColor()** изпраща специфична цветова команда към устройство, свързано чрез Bluetooth. Тази функция картографира имената на цветовете към съответните команди и ги предава към външно устройство, като например LED матрица.

Параметри:

- **color**: **String**, представляващ цвета, който трябва да бъде изпратен. Валидни стойности са: "red", "green", "blue", "white" и "black".
- **bluetoothManager**: Екземпляр на **BluetoothManager**, отговорен за управление на Bluetooth връзката и изпращане на командата за цвета.

Функционалност:

- Функцията проверява стойността на параметъра **color** и изпраща съответната команда към Bluetooth устройството:

- За **"red"**: Изпраща командата **"set-leds-red"**.
- За **"green"**: Изпраща командата **"set-leds-green"**.
- За **"blue"**: Изпраща командата **"set-leds-blue"**.
- За **"white"**: Изпраща командата **"set-leds-white"**.
- За **"black"**: Изпраща командата **"set-leds-black"**.

```
fun handshakeSendPixelQaurterRows(
    matrix: MutableState<RGBMatrix>,
    bluetoothManager: BluetoothManager,
    context: Context // Add context to show Toast messages
) { }

fun performHandshake(): Boolean { }
fun sendMatrixRows(): Boolean { }
fun terminateConnection() { }
```

Функцията **handshakeSendPixelQaurterRows()** управлява процеса на предаване на данни от пикселна мрежа към устройство, свързано чрез Bluetooth. Тя използва протокол за ръкостискане, изпраща данните ред по ред и прекратява връзката по подходящ начин. Също така осигурява обратна връзка към потребителя чрез логове и опционални Toast съобщения.

Параметри:

- **matrix**: **MutableState<RGBMatrix>**, представляваща данните за пикселната мрежа, които ще бъдат изпратени към Bluetooth устройството.
- **bluetoothManager**: Екземпляр на **BluetoothManager**, който управлява Bluetooth връзката и предаването на данни.
- **context**: **Context**, използван за показване на Toast съобщения, осигуряващи визуална обратна връзка за потребителя.

Функционалност:

- Опитва се да установи връзка чрез протокол за ръкостискане (handshake), изпращайки съобщение "syn" и очаквайки "syn-ack".
- Ако ръкостискането е успешно, предава данните от мрежата ред по ред, като всеки ред се разделя на четири части.
- Прекратява връзката с изпращане на съобщение "fin" и очаква "fin-ack".
- Ако предаването или ръкостискането се провалят, функцията уведомява потребителя за неуспех.

Функцията **performHandshake()** се опитва да установи връзка с Bluetooth устройство чрез протокол за ръкостискане.

Връща:

- **Boolean**: Връща true, ако ръкостискането е успешно, в противен случай връща false.

Функционалност:

- Изпраща съобщение "syn" и изчаква отговор "syn-ack" от Bluetooth устройството.
- Ако "syn-ack" бъде получен, изпраща съобщение "ack" и приключва процеса на ръкостискане.
- В случай на неуспех, функцията опитва повторно ръкостискане до три пъти, като дава обратна връзка чрез логове и Toast съобщения.

Функцията **sendMatrixRows()** отговаря за предаването на данните от пикселната мрежа ред по ред към Bluetooth устройството.

Връща:

- **Boolean**: Връща true, ако всички редове са успешно изпратени, в противен случай връща false.

Функционалност:

- Функцията преминава през всеки ред от матрицата, разделя го на четири части и предава всяка част последователно.
- За всяка част функцията изчаква потвърждение "ROW-SUCCESS" и повтаря изпращането до 20 пъти, ако потвърждението не бъде получено.
- Ако някоя част не бъде успешно предадена, функцията спира и връща false.

Функцията **terminateConnection()** прекратява връзката с Bluetooth устройството по правилен начин, използвайки протокол за прекратяване.

Функционалност:

- Изпраща съобщение "fin" и изчаква отговор "fin-ack" от устройството.
- Ако отговорът бъде получен, функцията прекратява връзката и логва успешното завършване.
- Ако не успее да прекрати връзката, опитва повторно до три пъти и дава обратна връзка чрез логове и Toast съобщения.

```
fun sendQuarterRow(
    matrix: MutableState<RGBMatrix>,
    row: Int,
    part: Int,
    bluetoothManager: BluetoothManager,
    addition: String = ""
) { }
```

Функцията **sendQuarterRow()** сериализира една четвърт от конкретен ред в пикселната мрежа и я изпраща към устройство, свързано чрез Bluetooth. Тя разделя реда на четвъртини, сериализира избраната част и я предава като едно съобщение.

Параметри:

- **matrix**: **MutableState<RGBMatrix>**, представляваща данните за пикселната мрежа, които ще бъдат изпратени.
- **row**: Цяло число, представляващо индекса на реда, който трябва да бъде изпратен.
- **part**: Цяло число, указващо коя четвърт от реда да бъде изпратена. Стойностите трябва да бъдат между 0 и 3, като всяка стойност съответства на конкретна четвърт от реда.
- **bluetoothManager**: Екземпляр на **BluetoothManager**, който управлява Bluetooth връзката и предаването на сериализираните данни.
- **addition**: **String**, която може да бъде добавена в началото на съобщението, съдържащо сериализираната четвърт от реда, преди предаването. Този параметър е опционален и по подразбиране е празен низ.

Функционалност:

- Функцията сериализира определената четвърт от реда, използвайки функцията **serializeQuarterRow**.
- След това конкатенира низа **addition** със сериализираните данни на четвъртината от реда, за да създаде пълното съобщение.
- Пълното съобщение се изпраща към Bluetooth устройството чрез функцията **bluetoothManager.sendData**.
- За отстраняване на грешки, функцията логва номера на реда, номера на частта и пълното съобщение.

```
fun serializeQuarterRow(
    matrix: MutableState<RGBMatrix>,
    row: Int = 0,
    part: Int = 0,
): String { }
```

Функцията **serializeQuarterRow()** сериализира определена четвърт от ред в пикселна мрежа във вид на хексадецимален низ. Тези сериализирани данни включват информация за цвета на пикселите и обикновено се използват за предаване към Bluetooth устройство.

Параметри:

- **matrix**: **MutableState<RGBMatrix>**, представляваща пикселните данни на мрежата, които трябва да бъдат сериализирани.
- **row**: Цяло число, представляващо индекса на реда, който трябва да бъде сериализиран. Стойността по подразбиране е 0.
- **part**: Цяло число, указващо коя четвърт от реда трябва да бъде сериализирана. Стойността по подразбиране е 0, като тя трябва да бъде между 0 и 3, всяка стойност съответстваща на конкретна четвърт от реда.

Връща:

- **String**: Връща хексадецимален низ, който представлява сериализираните данни за четвъртината от реда, включително информацията за цвета на пикселите и контролна сума.

Функционалност:

- Функцията създава **ByteArray** с размер 16, като всеки набор от четири байта представлява един пиксел от четвъртината на реда, която се сериализира.
- За всеки пиксел в определената четвърт, функцията изчислява позицията на пиксела в мрежата, конвертира стойностите на червено, зелено и синьо в байтове и ги съхранява в **ByteArray**.
- Функцията улавя всяко възникнало **IOException** и логва съобщение за грешка.
- **ByteArray** се конвертира в хексадецимален низ чрез **toHexString**.

- Контролна сума се добавя към сериализирания низ с помощта на функцията **addChecksumToRow**, за да се гарантира целостта на данните.

fun addChecksumToRow(data: String): String { }

Функцията **addChecksumToRow()** добавя контролна сума към даден хексадецимален низ. Контролната сума се изчислява от бинарното представяне на входните данни, за да се осигури целостта на данните по време на предаването.

Параметри:

- **data**: **String**, представляващ хексадецималните данни, към които ще бъде добавена контролната сума.

Връща:

- **String**: Връща оригиналния хексадецимален низ с добавена контролна сума в края.

Функционалност:

- Функцията конвертира входния хексадецимален низ в бинарното му представяне, използвайки функцията **hexStringToBinaryString**.
- Изчислява 8-битова контролна сума от бинарните данни с помощта на функцията **checkSum**.
- Контролната сума се конвертира от бинарен в хексадецимален низ с две символа, като се запълва с водещи нули, ако е необходимо.

- Оригиналният низ се конкатенира с контролната сума, и комбинираният низ се връща.

fun hexStringToBinaryString(hexString: String): String { }

Функцията **hexStringToBinaryString()** конвертира хексадецимален низ в неговия еквивалентен бинарен низ, където всяка хексадецимална цифра се преобразува в 4-битов бинарен код.

Параметри:

- **hexString**: **String**, представляващ хексадецималните данни, които трябва да бъдат конвертирани в бинарен низ.

Връща:

- **String**: Връща бинарен низ, еквивалентен на входния хексадецимален низ, където всяка хексадецимална цифра е преобразувана в 4-битов бинарен код.

Функционалност:

- Функцията преминава през всеки символ на входния **hexString**.
- За всеки символ добавя съответния 4-битов бинарен низ към променливата **binaryString**.
- Конверсията е чувствителна към големи и малки букви (например 'A' и 'a' се третират еднакво).
- Ако бъде срещнат невалиден хексадецимален символ, функцията хвърля **IllegalArgumentException**.

fun checkSum(data: String, blockSize: Int): String { }

Функцията **checkSum()** изчислява контролната сума на даден бинарен низ, като го разделя на блокове, изпълнява бинарно събиране

на всеки блок и след това прилага допълнение до единица върху резултата. Тази контролна сума обикновено се използва за проверка на целостта на данните по време на предаване.

Параметри:

- **data**: **String**, представляващ бинарните данни, за които ще се изчисли контролната сума.
- **blockSize**: **Int**, представляващ размера на блоковете, на които ще се раздели бинарният низ за изчисляване на контролната сума.

Връща:

- **String**: Връща допълнението до единица на резултата от бинарното събиране като бинарен низ, който служи като контролна сума.

Функционалност:

- Функцията допълва входния бинарен низ с водещи нули, ако неговата дължина не е кратна на **blockSize**.
- Инициализира резултата с първия блок от допълнените данни.
- След това функцията обхожда останалите блокове и изпълнява бинарно събиране на всеки блок с натрупания резултат.
- След обработка на всички блокове функцията изчислява допълнението до единица (**one's complement**) на резултата, което се връща като контролна сума.

```
fun onesComplement(data: String): String { }
```

Функцията **onesComplement()** изчислява допълнението до единица на даден бинарен низ. Допълнението до единица се получава чрез инвертиране на всеки бит в низа.

Параметри:

- **data**: **String**, представляващ бинарните данни, за които ще се изчисли допълнението до единица.

Връща:

- **String**: Връща нов бинарен низ, където всеки бит е инвертиран: '0' става '1', а '1' става '0'.

Функционалност:

- Функцията обхожда всеки символ в входния низ.
- За всеки символ инвертира бита: '0' се променя на '1', а '1' се променя на '0'.
- Инвертираните битове се съхраняват в **CharArray**, който след това се конвертира обратно в низ и се връща.

6.2. Проектиране на системния софтуер

Програмният код на Arduino проектите започва с две ключови функции: **setup**() и **loop**(). Те представляват основата на всяка Arduino програма и определят как микроконтролерът ще изпълнява своята логика. За разлика от други програмни среди, при които изпълнението на кода е линейно, в Arduino средата **setup**() функцията се извиква еднократно при стартиране на устройството и се използва за инициализация на хардуерни компоненти и настройки на променливи. След това функцията **loop**() се изпълнява непрекъснато, което осигурява постоянен цикъл за обработка на входни данни и управление на изходните устройства. Този цикличен модел е централен за начина, по който микроконтролерът реагира в реално време на промените в околната среда.

```
#include <SoftwareSerial.h>
#include <AltSoftSerial.h>
#include <FastLED.h>
#include "checksumbin.h"
```

```
#define MATRIX_SIZE 16
#define LEDS_DATA_PIN 12
#define NUM_LEDS 256
#define BLOCK_SIZE 8
#define SYN "syn"
#define SYN_ACK "syn-ack"
#define ACK "ack"
#define ROW_SUCCESS "ROW-SUCCESS"
#define ROW_FAIL "ROW-FAIL"
#define FIN "fin"
#define FIN_ACK "fin-ack"
#define LEDS_BLACK "set-leds-black"
#define LEDS_WHITE "set-leds-white"
#define LEDS_RED "set-leds-red"
#define LEDS_GREEN "set-leds-green"
#define LEDS_BLUE "set-leds-blue"
#define QUARTER_ROW_HEX_CHAR_SIZE 34
#define QUARTER_ROW_BINARY_CHAR_SIZE 136

SoftwareSerial bluetoothManager(9, 10);  // RX | TX
CRGB leds[NUM_LEDS];
char incomingMessage[QUARTER_ROW_HEX_CHAR_SIZE + 8];
uint8_t messageIndex = 0;
```

Този сегмент от кода съдържа дефиниции на макроси и променливи, които са от съществено значение за функционирането на Arduino проекта, включващ управление на LED матрица чрез Bluetooth.

Библиотеки:

- **SoftwareSerial**: Библиотека за създаване на софтуерен сериен интерфейс, използван за комуникация с Bluetooth.
- **AltSoftSerial**: Подобрена версия на SoftwareSerial за едновременно използване на други комуникационни интерфейси.
- **FastLED**: Библиотека за управление на LED ленти и матрици.
- **checksumbin.h**: Външен хедър файл, който вероятно съдържа функции за изчисляване на контролни суми.

Макроси:

- **MATRIX_SIZE**: Определя размера на LED матрицата – 16x16.
- **LEDS_DATA_PIN**: Пин 12 е зададен като изход за данните към LED матрицата.
- **NUM_LEDS**: Общият брой на LED светодиодите в матрицата (16x16 = 256).
- **BLOCK_SIZE**: Определя размера на блоковете при обработката на данните, използван за разделяне на редове или колони.
- **SYN**, **SYN_ACK**, **ACK**: Символни низове, използвани за инициализация на комуникация чрез протокола за ръкостискане.
- **ROW_SUCCESS**, **ROW_FAIL**: Символни низове, използвани за указване на успешното или неуспешното изпращане на ред.
- **FIN**, **FIN_ACK**: Команди за завършване на комуникацията чрез протокола за прекратяване на връзката.
- **LEDS_BLACK**, **LEDS_WHITE**, **LEDS_RED**, **LEDS_GREEN**, **LEDS_BLUE**: Низове, които се използват за настройка на LED матрицата на определен цвят (черен, бял, червен, зелен, син).
- **QUARTER_ROW_HEX_CHAR_SIZE**: Размерът на един сериализиран четвърт ред в хексадецимален формат – 34 символа.
- **QUARTER_ROW_BINARY_CHAR_SIZE**: Размерът на един сериализиран четвърт ред в двоичен формат – 136 символа.

Променливи:

- **bluetoothManager**: Създава софтуерен сериен интерфейс за Bluetooth комуникация на пинове 9 (RX) и 10 (TX).
- **leds**: Масив от **CRGB** обекти, представляващ състоянието на всички LED светодиоди в матрицата.
- **incomingMessage**: Масив от символи, който съдържа входящите съобщения през Bluetooth, с размер **QUARTER_ROW_HEX_CHAR_SIZE + 8** за обработка на допълнителни данни.
- **messageIndex**: Цяло число, което се използва за проследяване на индекса на текущото съобщение, получено през Bluetooth.

void setup() { }

Функцията **setup()** инициализира сериен интерфейс, Bluetooth модул и LED лента. Тя конфигурира основните настройки и подготвя системата за работа.

Функционалност:

- Инициализира серийната комуникация за дебъгинг и мониторинг със скорост на предаване от **9600** бодове, използвайки **Serial.begin(9600)**.
- Инициализира Bluetooth комуникацията чрез **SoftwareSerial** на пинове 9 (RX) и 10 (TX), също със скорост от **9600** бодове, използвайки **bluetoothManager.begin(9600)**.
- Задава буфера **incomingMessage** като празен низ, инициализирайки го с \0 (терминиращ символ).
- Настройва LED лентата с помощта на библиотеката **FastLED**, като определя типа LED (WS2812B), пина за данни (**LEDS_DATA_PIN**) и подредбата на цветовете (**GRB**). Масивът **leds** съхранява състоянието на всички светодиоди, а общият брой е **NUM_LEDS** (256 светодиода).

void loop() { }

Функцията **loop()** е основната функция, която се изпълнява непрекъснато, обработвайки входящите съобщения от Bluetooth модула. Тя следи за налични данни, събира входящите символи и предизвиква подходящи действия в зависимост от получените съобщения.

Функционалност:

- Постоянно проверява дали има налични данни от Bluetooth модула чрез **bluetoothManager.available()**.
- Чете всеки символ от входящия поток на Bluetooth чрез **bluetoothManager.read()**, като събира символите в буфера **incomingMessage**, докато не бъде срещнат символ за нов ред (\n) или връщане на каретката (\r).
- След като съобщението е напълно получено (когато бъде срещнат край на реда), то се предава на функцията **processMessage** за допълнителна обработка.
- След обработването на съобщението, буферът **incomingMessage** се изчиства чрез **memset**, а индексът **messageIndex** се нулира, за да бъде готов за следващото съобщение.

Структура:

Проверка за налични данни: Ако има налични данни от Bluetooth модула, се чете символ по символ.

Събиране на съобщение: Всеки символ се добавя към **incomingMessage**, докато не бъде достигнат символ за нов ред.

Обработка на съобщение: След като съобщението е пълно, се предава на **processMessage**.

Ресет на буфера: След всяко съобщение буферът се изчиства и индексът се нулира.

```
void processMessage(char* message) { }
```

Функцията processMessage() обработва и интерпретира входящите съобщения, получени чрез Bluetooth. Тя разпознава различни команди и извършва съответните действия, като изпращане на потвърждения или управление на светодиодите.

Параметри:

- message: Указател към низ (char*), представляващ съобщението, получено чрез Bluetooth, което се обработва с цел да предизвика различни действия.

Функционалност:

- Функцията интерпретира и обработва предварително дефинирани съобщения като SYN, SYN_ACK, ACK, FIN и команди за управление на светодиодите.
- Изпраща съответните отговори обратно чрез Bluetooth, като SYN-ACK, ACK, ROW_SUCCESS, ROW_FAIL и FIN_ACK.
- Обработва данни за пиксели, които са с префикс "data:", и ги проверява с помощта на контролна сума. Ако данните са валидни, се актуализира LED дисплеят.
- Управлява цветовете на светодиодите според специфични команди, като задава светодиодите да светят в черно, бяло, червено, зелено или синьо.
- Ако съобщението е непознато, функцията го отпечатва чрез Bluetooth за целите на дебъгване.

void setLedsColor(CRGB color) { }

Функцията setLedsColor() задава на целия LED лента определен цвят и показва резултата.

Параметри:

- color: Стойност от тип CRGB, представляваща цвета, който ще бъде зададен на всички светодиоди в лентата.

Функционалност:

- Задава всички светодиоди в лентата да светят в посочения color.
- Използва функцията FastLED.show(), за да покаже зададения цвят на светодиодите незабавно.
- Добавя кратко забавяне, за да гарантира, че цветът е зададен коректно.

bool checkCheckSum(char message) { }*

Функцията checkCheckSum() проверява целостта на съобщение чрез конвертиране в бинарен формат, изчисляване на контролна сума и сравнение на резултата, за да определи дали съобщението е валидно.

Параметри:

- message: Указател към низ (char*), представляващ хексадецималния низ, който трябва да бъде проверен.

Връща:

- bool: Връща true, ако контролната сума е валидна и съобщението е правилно формирано; в противен случай връща false.

Функционалност:

- Конвертира хексадецималното съобщение в бинарен низ чрез функцията hexData_to_binaryData.
- Изчислява контролната сума на бинарните данни, използвайки функцията checkSum.
- Сравнява резултата от контролната сума с очакваната стойност чрез функцията result_checker.
- Ако контролната сума е валидна, обработва четвъртинка от данните и връща true; ако не е валидна, връща false.

void hexData_to_binaryData(const char hexData, char* binaryData) { }*

Функцията hexData_to_binaryData() конвертира хексадецимален низ в неговия еквивалентен бинарен низ, където всяка хексадецимална цифра се преобразува в 4-битов бинарен код.

Параметри:

- hexData: Константен указател към низ (const char*), представляващ хексадецималните данни, които трябва да бъдат конвертирани в бинарен формат.
- binaryData: Указател към низ (char*), където ще бъде съхранен резултантният бинарен низ. Този низ се конструира чрез добавяне на 4-битовия бинарен еквивалент на всяка хексадецимална цифра.

Функционалност:

- Инициализира променливата binaryData като празен низ.
- Обхожда всяка хексадецимална цифра от hexData и добавя съответния 4-битов бинарен низ към binaryData.
- Функцията разпознава както малки, така и големи букви за хексадецималните цифри.
- Ако бъде срещнат невалиден хексадецимален символ, функцията добавя "????" към binaryData, за да посочи грешка.

void checkSum(char data, int block_size, char* result) { }*

Функцията checkSum() изчислява контролната сума на бинарен низ, като го разделя на блокове с определен размер, извършва бинарно събиране на всеки блок и прилага допълнение до едно (one's complement). Контролната сума се използва за осигуряване на целостта на данните по време на предаването.

Параметри:

- data: Указател към низ (char*), представляващ бинарния низ, за който ще се изчислява контролната сума. Низът може да бъде допълнен с водещи нули, за да съответства на размера на блока.
- block_size: Цяло число (int), указващо размера на всеки блок (в брой битове), използван при изчисляването на контролната сума.
- result: Указател към низ (char*), където ще бъде съхранена изчислената контролна сума. Контролната сума е допълнението до едно на резултата от бинарното събиране.

Функционалност:

- Функцията проверява дали дължината на data е делима на block_size. Ако не е, низът се допълва с водещи нули.
- Инициализира променливата result с първия блок от данните.
- Извършва поетапно бинарно събиране на блоковете от данни, като актуализира резултата след всяко събиране.
- Ако по време на събирането се генерира пренос (carry), той се обработва и се пропагира правилно.
- Накрая, прилага се допълнение до едно (one's complement) на резултата, като това представлява контролната сума на входните данни.

```
void Ones_complement(char* data) { }
```

Функцията Ones_complement() изчислява допълнение до едно (one's complement) на даден бинарен низ, като инвертира всеки бит – променя '0' на '1' и '1' на '0'.

Параметри:

- data: Указател към низ (char*), представляващ бинарния низ, за който ще се изчислява допълнението до едно. Низът се модифицира на място.

Функционалност:

- Функцията обхожда всеки символ на низа data.
- За всеки символ проверява дали е '0' или '1' и го инвертира.
- Модифицираният низ представлява допълнението до едно на оригиналния вход, като инверсията се извършва директно в оригиналния низ.

bool result_checker(char result, int block_size) { }*

Функцията result_checker() проверява дали дадена контролна сума е валидна, като проверява дали всички битове в резултата са нули.

Параметри:

- result: Указател към низ (char*), представляващ бинарния резултат на контролната сума, който трябва да бъде проверен.
- block_size: Цяло число (int), указващо размера на блока (брой битове), който трябва да бъде проверен.

Връща:

- bool: Връща true, ако всички битове в result са '0', което означава, че контролната сума е валидна. В противен случай връща false.

Функционалност:

- Функцията обхожда всеки бит в низа result.
- Ако бъде намерен бит, различен от нула, функцията връща false.
- Ако всички битове са '0', функцията връща true, което означава, че контролната сума е валидна и целостта на данните е запазена.

void processQuarterRow(const char rowData) { }*

Функцията processQuarterRow() обработва четвърт от ред с пикселни данни, като преминава през всеки пиксел в тази част от реда и предава данните за пиксела на функцията processPixel.

Параметри:

- rowData: Константен указател към низ (const char*), представляващ бинарните данни за четвърт от ред с пиксели.

Функционалност:

- Функцията обхожда всеки пиксел в четвърт от реда (предполага се, че има 4 пиксела на четвърт ред).
- За всеки пиксел съответният сегмент от rowData се предава на функцията processPixel за обработка.
- Тази функция се използва за актуализиране на LED лентата с цветните данни за част от реда.

```
void processPixel(const char* pixelData) { }
```

Функцията processPixel() обработва данните за отделен пиксел, като извлича информацията за реда, колоната и RGB цвета от бинарните данни и задава съответния светодиод на указаните координати с посочения цвят.

Параметри:

- pixelData: Константен указател към низ (const char*), представляващ бинарните данни за един пиксел, включително неговата позиция (ред и колона) и информация за цвета (RGB).

Функционалност:

- Извлича номера на реда и колоната от първите 8 бита на pixelData (по 4 бита за ред и колона).

- Извлича компонентите на цвета – червен (R), зелен (G) и син (B) – от следващите 24 бита на pixelData (по 8 бита за всеки компонент).
- Изчислява правилния индекс за светодиода в LED лентата въз основа на номера на реда и колоната, като отчита зигзаговия модел на подредба на светодиодите.
- Задава цвета на светодиода на изчисления индекс, използвайки извлечените стойности за RGB.

uint8_t binaryToDecimal(const char binaryString, size_t length) { }*

Функцията binaryToDecimal() конвертира бинарен низ в неговия десетичен еквивалент. Конверсията се ограничава до размера, зададен от параметъра length.

Параметри:

- binaryString: Константен указател към низ (const char*), представляващ бинарния низ, който трябва да бъде конвертиран в десетична стойност.
- length: Параметър от тип size_t, указващ дължината на бинарния низ, която трябва да бъде обработена при конверсията.

Връща:

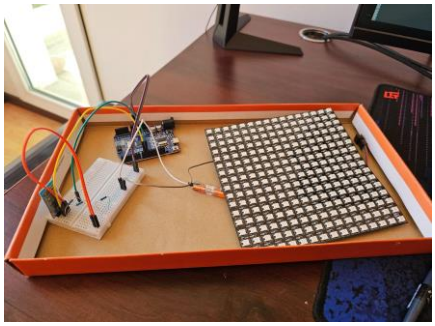
- uint8_t: Връща десетичния еквивалент на входния бинарен низ.

Функционалност:

- Функцията преминава през бинарния низ отляво надясно, като за всяка позиция измества резултата наляво и добавя съответната стойност на бита.
- Обработва само указаната дължина (length) на бинарния низ и я конвертира в десетично число.

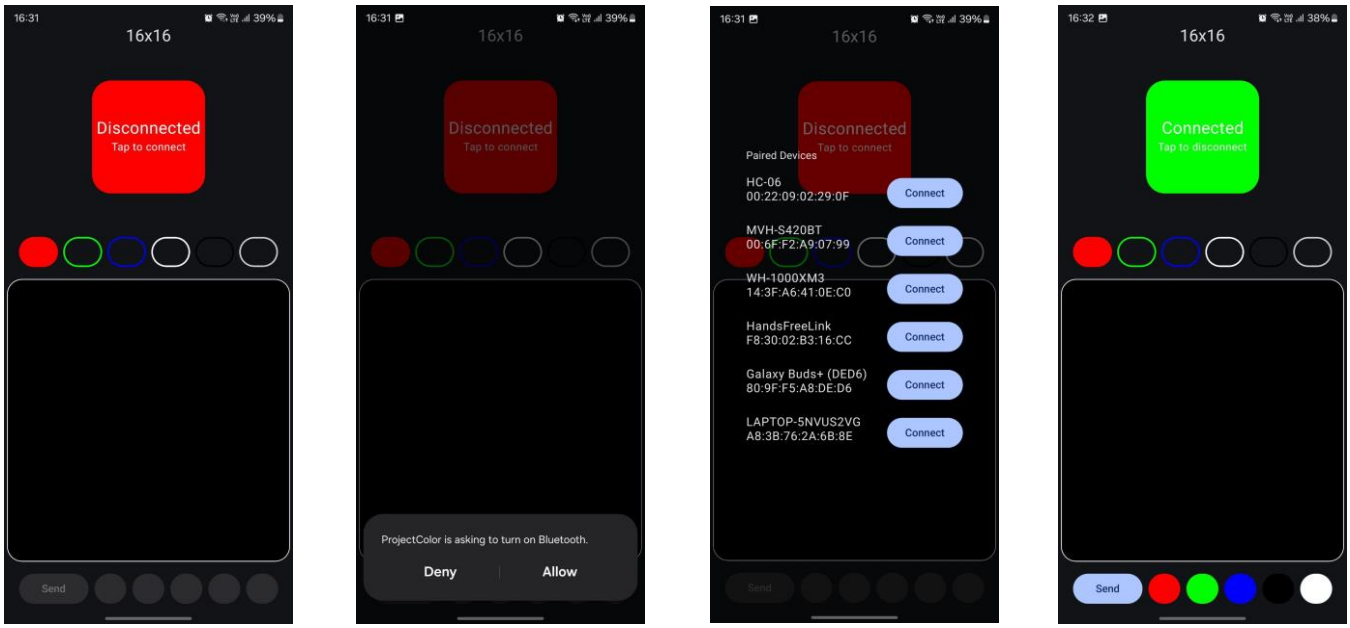
7. Опитни резултати

Целта на експериментите е да се тества функционалността и ефикасността на приложния софтуер и на системния софтуер и тяхната комуникация. Основния фокус е върху възпроизвеждането на правилна комуникация, както и възможните грешки.



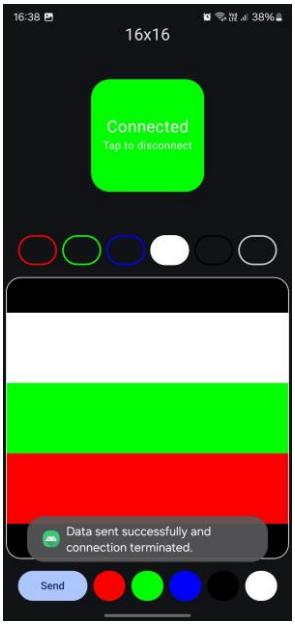
Фиг. 7.1 Снимка на проекта (платките)

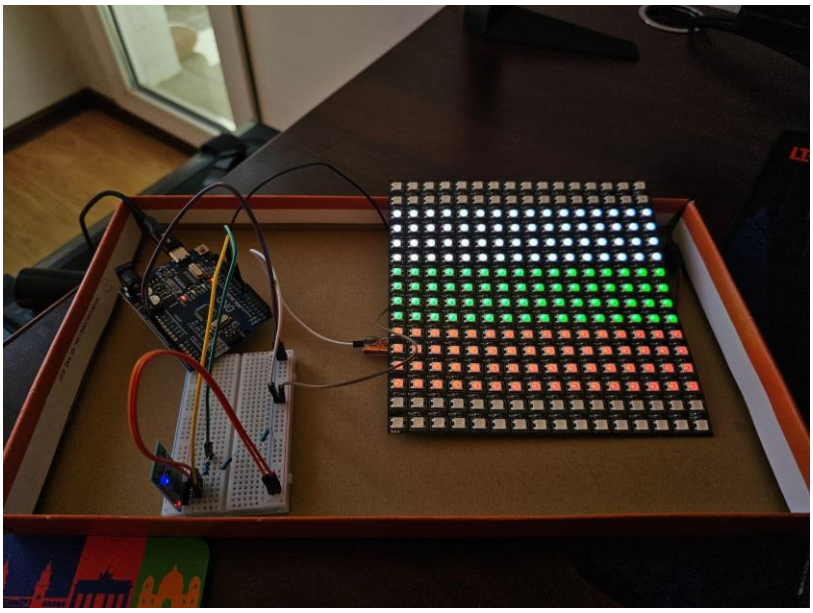
В долните екранни снимки се разглежда приложението с неговите функции (главен екран, диалог за включване на блутут, списък със сдвоени устройства, успешно свързване с устройство).



Фиг. 7.2-7.5 Снимки на приложението

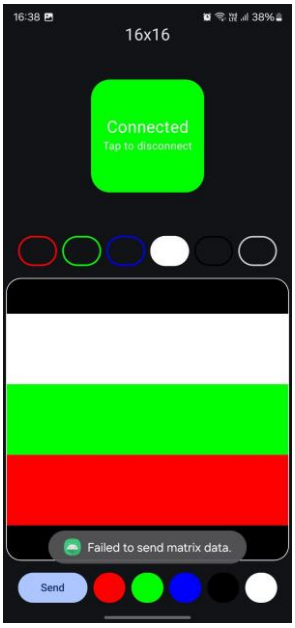
В долните снимки е показано успешно изпращане на картинка, в случая визуализирано българското знаме.

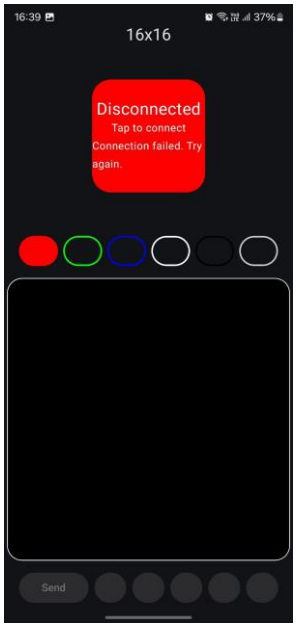




Фиг. 7.6-7.7 Снимки на успешна комуникация

В долните екранни снимки са демонстрира на неуспешно изпращане на картинка, както и неуспешно свързване към устройство.





Фиг. 7.8 Снимка на неуспешна комуникация

8. Заключение

Дипломната работа, озаглавена “Система за управление на LED матрица с Bluetooth свързаност,” постигна основната си цел – разработването на интегрирана система, която позволява безжично управление на LED матрица чрез ефективен обмен на данни между мобилно устройство и хардуерна система. Проектът демонстрира успешна реализация на система, която съчетава стабилността и функционалността на Arduino Uno R3, леснотата на използване и модерността на Kotlin за Android приложението, както и гъвкавостта на Bluetooth технологията за безжично свързване.

В хода на разработката бяха изпълнени три основни задачи. Първо, създаден бе интуитивен потребителски интерфейс, който улеснява управлението на LED матрицата и прави взаимодействието с устройството лесно и достъпно за потребителя. Второ, разработен бе алгоритъм, който осигурява надеждно предаване на данни в безжичен режим, гарантирайки стабилност дори при наличие на смущения. Трето, системата беше успешно интегрирана на хардуерно ниво, което осигурява безпроблемно визуализиране на промените на LED матрицата въз основа на изпратените команди.

Резултатите от проекта показват, че е възможно да се разработи рентабилно и функционално решение за безжично управление на LED матрица с помощта на достъпни технологии. Платформата Arduino Uno R3 и езикът Kotlin за мобилната апликация доказаха своята съвместимост и надеждност, като осигуриха стабилна работа на системата. Използването на Bluetooth технологията допълнително затвърди успеха на проекта, като осигури надеждно безжично свързване.

Тази система притежава потенциал за бъдещи подобрения и разширения. Например, добавянето на нови функционалности към мобилното приложение, разширяване на възможностите на LED матрицата или интегриране на други видове безжична комуникация могат да повишат гъвкавостта и приложимостта на системата в

различни индустриални и потребителски сценарии. Допълнително, проектът може да бъде адаптиран за по-сложни LED системи и по-големи дисплеи, като предложи още повече възможности за персонализация и контрол.

Заключително, този проект показа не само техническите възможности на съвременните хардуерни и софтуерни технологии, но също така отвори врати за бъдещи иновации и подобрения в сферата на безжичното управление на различни системи. Системата за управление на LED матрица с Bluetooth свързаност може да бъде развита и адаптирана за множество приложения, което я прави гъвкаво и ценно решение в контекста на модерните технологии.

9. Използвана литература

1. Microchip. "ATmega328P Datasheet." https://www.microchip.com
2. Arduino. "Arduino Uno Rev3." https://store.arduino.cc/products/arduino-uno-rev3
3. Microchip. "AVR Instruction Set." https://www.microchip.com/en-us/product/ATmega328P
4. Arduino. "Arduino Uno R3 Overview." https://www.arduino.cc/en/Main/ArduinoBoardUno
5. Adafruit. "Arduino Uno R3 FAQ." https://learn.adafruit.com/arduino-tips-tricks-and-techniques/arduino-uno-faq
6. Arduino IDE Documentation. https://docs.arduino.cc/software/ide-v1
7. Android Developers Guide. https://developer.android.com/guide
8. Google. "Android Architecture Components." https://developer.android.com/topic/libraries/architecture
9. Google. "Bluetooth Overview in Android." https://developer.android.com/guide/topics/connectivity/bluetooth
10. Android SDK Documentation. https://developer.android.com/studio
11. JetBrains. "Kotlin Programming Language." https://kotlinlang.org
12. Google Developers. "Kotlin and Android Development." https://developer.android.com/kotlin
13. Android Developers. "Kotlin Coroutines Overview." https://developer.android.com/kotlin/coroutines
14. Nordic Semiconductor. "Bluetooth Low Energy Overview." https://www.nordicsemi.com/Products/Low-power-short-range-wireless/Bluetooth-Low-Energy
15. Rajguru Electronics. “HC06 Core Bluetooth Module” https://rajguruelectronics.com/Product/707/HC-06%20core%20bluetooth%20module.pdf
16. Arduino. “Arduino Uno R3 Schematic PDF” https://www.arduino.cc/en/uploads/Main/Arduino_Uno_Rev3-schematic.pdf

10. Анотация

Дипломната работа, озаглавена “Система за управление на LED матрица с Bluetooth свързаност,” представя разработването на система, която осигурява безжично управление на LED матрица чрез използване на Bluetooth технология. Основната цел на проекта е да създаде ефективно и надеждно решение за обмен на данни между мобилно устройство и хардуерната система.

В процеса на работа са изпълнени три основни задачи: проектиране на интуитивен потребителски интерфейс за мобилно приложение, разработване на алгоритъм за стабилно безжично предаване на данни, и интегриране на хардуера с LED матрицата за правилно визуализиране на командите. Платформата Arduino Uno R3 е избрана за хардуерната част, а мобилното приложение е разработено с помощта на Kotlin за Android.

Резултатите от проекта показват, че разработената система работи надеждно и предлага рентабилно решение за безжично управление на LED дисплеи. Използването на Kotlin за разработване на приложението допълнително улеснява потребителския достъп и управление на системата. Системата притежава потенциал за бъдещо разширение и подобрения, включително интегриране на нови функционалности, разширяване на капацитета на LED матрицата и използването на други безжични технологии. Тя може да бъде приложена в различни области като домашна автоматизация, индустриални решения и образователни проекти.