

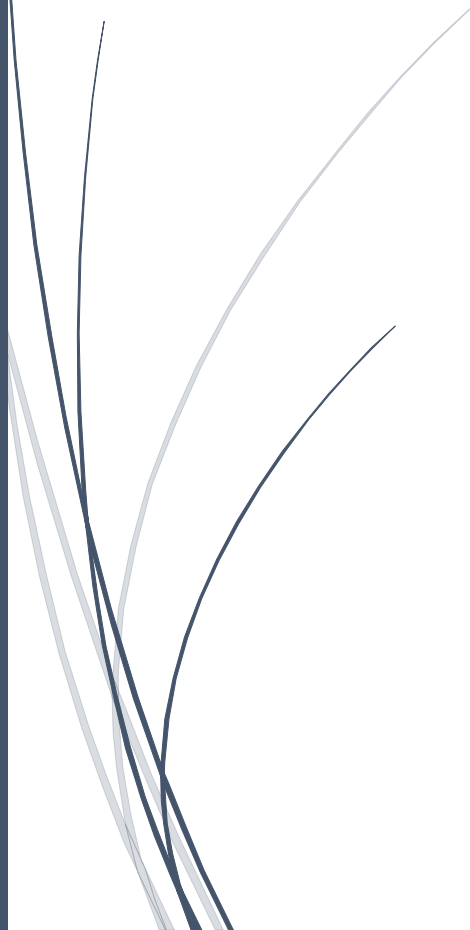


2019-06-12

Zamek RFID

Systemy wbudowane

Autorzy:

- Jędrzej Dobrucki (lider) 216748
 - Kamil Celejewski 216733
 - Maciej Bartos 216719
- 

Spis treści

1.	Projekt RFID.....	
1.1.	Wstęp.....	
1.2.	Funkcjonalności systemu.....	
1.3.	Wykorzystane urządzenia.....	
2.	Dokumentacja użytkowania.....	
2.1.	Instrukcja użytkowania.....	
3.	Dokumentacja techniczna.....	
3.1.	SPI.....	
3.2.	Czytnik kart RFID.....	
3.3.	TIMER.....	
3.4.	GPIO (Dioda oraz guziki).....	
3.5.	Obsługa przycisków z tłumieniem drgań.....	
3.6.	PWM + Buzzer.....	
3.7.	Schemat układu elektronicznego.....	
4.	Analiza skutków awarii.....	
5.	Źródła	

1. Projekt RFID

1.1 Wstęp

Celem projektu było stworzenie elektronicznego zamka, który można otwierać za pomocą czytnika RFID. Do wykonania tego zadania wykorzystaliśmy płytke **LPC 2138**.

1.2 Funkcjonalności systemu

Poniższa tabela przedstawia wykonane przez nas funkcjonalności ich realizację (czy zostały wykonane) oraz osoby odpowiedzialne za wykonanie danej części.

Funkcjonalności	Realizacja	Osoba odpowiedzialna
PWM	Wykonano	Maciej Bartos
TIMER	Wykonano	Jędrzej Dobrucki
RFID	Wykonano	Kamil Celejewski
GPIO	Wykonano	Maciej Bartos
Buzzer	Wykonano	Maciej Bartos
Eliminacja drgań przycisków	Wykonano	Jędrzej Dobrucki
SPI	Wykonano	Kamil Celejewski
Układ dioda + rezystor	Wykonano	Jędrzej Dobrucki

1.3 Wykorzystane urządzenia

- Płytko LPC 2138
- Czytnik kart RFID-RC522
- Moduł z buzzerem pasywnym bez generatora
- Dioda LED
- Rezystor

1.3 Udział procentowy członków grupy w projekcie:

- Jędrzej Dobrucki – 34%
- Kamil Celejewski – 33%
- Maciej Bartos – 33%

2. Dokumentacja użytkowania

2.1 Instrukcja użytkowania

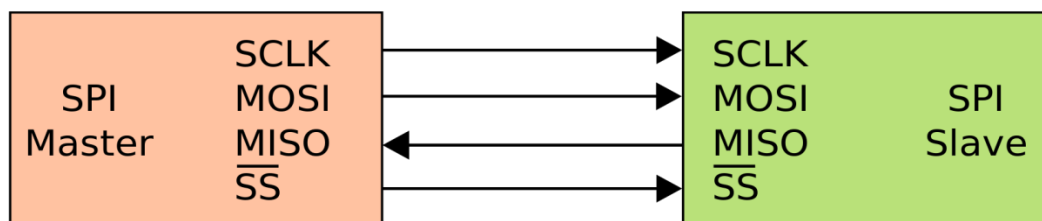
Program umożliwia otworenie zamka przy pomocy czytnika RFID i identyfikatora w postaci karty plastikowej. Po przyłożeniu karty do czytnika jeśli ma ona poprawny numer zaświeci się dioda i zostanie odegrany ton za pomocą buzzera w innym wypadku system nie zareaguje. Nasz system posiada dodatkowy sposób otwierania zamka, który ma zastąpić czytnik jeśli wystąpi jego ewentualna awaria. Za pomocą przycisków wciśniętych w odpowiedniej kolejności można otworzyć zamek. W naszym systemie zamek otwarty rozumiany jest jako zapalona dioda.

3. Dokumentacja techniczna

3.1 SPI

3.1.1 Interfejs SPI

Szeregowy interfejs SPI (Serial Peripheral Interface) służy do dwukierunkowej, synchronicznej transmisji danych pomiędzy mikrokontrolerem, a zewnętrznymi układami peryferyjnymi. Interfejs SPI posiada trzy przewody, składa się z dwóch linii przesyłających dane w różnych kierunkach oraz linii z sygnałem taktującym synchronizującym transfer danych. Transfer danych odbywa się w układzie master-slave.



- **SCK** (Serial CLock) - zegar synchronizujący transmisj
- **MOSI** (Master Output Slave Input) – dane od master do slave
- **MISO** (Master Input Slave Output) – dane od jednostki podporządkowanej do nadrzędnej
- **SS** (Slave Select) – służy do wybierania poszczególnych układów podrzędnych

Sygnał taktujący jest nadawany zawsze przez urządzenie master bez względu na to czy dane są przesyłane do niego czy wysyłane do slave. MISO pozwala wysyłać dane do urządzenia master ze slave a MOSI odwrotnie. Linia SCK jest służy do synchronizacji transmisji. SS pozwala na wybieranie układu podrzędnego poprzez ustawienie odpowiedniego stanu wysokiego lub niskiego w zależności od urządzenia. Nadawanie na linii MOSI wiąże się zawsze z odbieraniem danych na MISO niekoniecznie mających sens.

3.1.2 Rejestry SPI

W programie zostały użyte cztery rejestry SPI:

- **SPCR** - służy do konfiguracji interfejsu i sterowania jego pracą
- **SPDR** - służy do wysłania oraz do odczytu bajtów danych
- **SPSR** - zawiera flagi sygnalizujące stan interfejsu
- **SPCCR** - kontroluje częstotliwość SCK

3.1.3 Funkcje

Program korzysta z SPI do komunikacji z czytnikiem RFID.

- `void SPI_Master_Init()` – inicjalizuje SPI
- `void SPI_Master_Write(tU8 data)` – wysyła jeden bajt do slave
- `tU8 SPI_Master_Read()` – odczytuje jeden bajt od slave

void SPI_Master_Init():

1. Wybieram piny P0.4, P0.5, P0.6, P0.7 by służyły jako SCK0, MISO0, MOSI0 i SS.

```
PINSEL0 = PINSEL0 | 0x00001500;
```

2. Ustawiamy stan pinu 0.7 na wysoki aby połączony chip został automatycznie odłączony.

```
IOSET |= (1 << 7);
```

3. Ustawiamy pin 0.7 jako output.

```
IODIR |= (1 << 7);
```

4. Konfiguracja rejestrów SPCR i SPCCR.

Ustawienie piątego bitu na jeden powoduje że SPI pracuje w trybie master, w którym posiada pełną kontrolę nad sesją komunikacyjną.

```
SPI_SPCR = 0x0020;
```

5. Wartość minimalna rejestru SPCCR to 8 i można w nim zapisywać tylko parzyste wartości.
Rejestr kontroluje częstotliwość master SCK.
(PCLK / SPCCR0 = SPI_RATE).
SPI_SPCCR = 0x08;

6. Ustawienie SCK I MOSI jako output.

```
IODIR |= (1 << 4 | 1 << 6)
```

void SPI_Master_Write(tU8 data):

1. Zmienna na odebrany flush byte.

```
char flush;
```

2. Wpisujemy do rejestry SPDR byte , który chcemy wysłać.

```
SPI_SPDR = data;
```

3. Oczekiwanie na zakończenie transmisji i odczytanie flush byte z SPDR. Zakończeniu transmisji jest sygnalizowane ustawieniem 7 bitu rejestru SPSR na 1.

```
while ( (SPI_SPSR & 0x80) == 0 );  
flush = SPI_SPDR;
```

tU8 SPI_Master_Read():

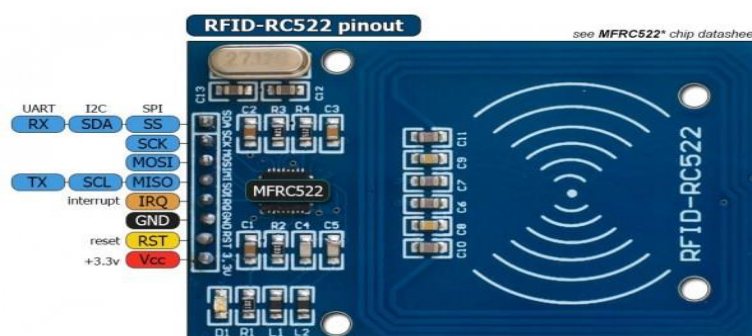
1. Wysłanie flush byte.

```
SPI_SPDR = 0xFF;
```

2. Oczekiwanie na zakończenie transmisji i zwrócenie wartości odebranej.

```
while ( (SPI_SPSR & 0x80) == 0 );  
return SPI_SPDR;
```

3.2 Czytnik RFID



W projekcie został wykorzystany czytnik RFID model RC522, z którym komunikacja odbywa się za pomocą SPI. Wymiana informacji z urządzeniem polega na wysyłaniu odpowiednich wartości do konkretnych rejestrów oraz ich odczycie. Odczyt danych z karty MIFARE odbywa się za pośrednictwem wbudowanej kolejki FIFO, do której należy wysłać odpowiednie rozkazy by uzyskać żądane wartości od MIFARE.

Zapis wartości do rejestru polega na wysłaniu do urządzenia 2 bajtów. Pierwszy bajt odpowiada adresowi rejestru, do którego chcemy wysłać pewną wartość, a drugi odpowiada tej wartości. Adres zapisywany jest na 8 bitach w formacie 0XXXXXX0 z czego pierwszy bit musi być 0 jeśli ma zostać odczytana wartość rejestru a jeśli chcemy zapisać daną wartość do rejestru wtedy adres przyjmuje postać 1XXXXXX0.

Odczyt ID karty polega na przeczytaniu 16 bitów z pierwszego sektora karty, który zawiera również informacje o producencie karty. Pierwsze 4 bajty, które odczytamy są odwróconym numerem ID zapisanym szesnastkowo. Karty MIFARE posiadają po 16 sektorów do zapisu i odczytu danych dlatego można na nich zapisywać np.: ID konkretnego użytkownika karty i jego nazwę by zwiększyć bezpieczeństwo. W naszym systemie sprawdzenie czy MIFARE jest upoważniony do otworzenia zamka polega na sprawdzaniu czy 4 pierwsze bajty są zgodne z tymi zapisanymi w kodzie.

3.2.1 Rejestry RC522

Rejestry wykorzystane w programie:

- **CommandReg** - wykonuje komendy zapisane na bitach [3-0].
- **CommIEReg** - decyduje o tym, które przerwania będą obsługiwane.
- **CommIrqReg** - informacje o przerwaniach.
- **ErrorReg** - pokazuje błędy ostatnio wykonanej komendy.
- **FIFODataReg** - 64 bitowy bufor.
- **FIFOLevelReg** - ilość bitów przechowywana w rejestrze bufora.
- **ControlReg** - miscellaneous control registers.
- **BitFramingReg** - adjustments for bit-oriented frames.
- **TModeReg** - definiuje ustawienia wewnętrznego timer.
- **TPrescalerReg** - definiuje ustawienia wewnętrznego timer.
- **TReloadRegH** - definiuje 16 bitową wartość odświeżenia timer.
- **TReloadRegL** - definiuje 16 bitową wartość odświeżenia timer.
- **VersionReg** - zawiera wersję software.
- **ModeReg** - definiuje model transmisji i odbierania.
- **TxControlReg** - kontroluje antenę.
- **TxAutoReg** – (TxASKReg) kontroluje ustawienia modulacji transmisji.

3.2.2 Komendy MIFARE

- PICC_REQIDL 0x26
- PICC_ANTICOLL 0x93

PICC_REQIDL - Invites PICCs in state IDLE to go to READY

PICC_ANTICOLL- przeciw działanie kolizji.

3.2.3 Funkcje

Lista funkcji wykorzystywanych do obsługi czytnika RFID:

- void writeMFRC522(tU8 addr, tU8 val);
- tU8 readMFRC522(tU8 addr);
- void reset();
- void setBitMask(tU8 reg, tU8 mask);
- void clearBitMask(tU8 reg, tU8 mask);
- void antennaOn();
- tU8 MFRC522ToCard(tU8 command, tU8 *sendData, tU8 sendLen, tU8 *backData, tU32 *backLen);
- tU8 MFRC522Request(tU8 reqMode, tU8 *TagType);
- tU8 anticoll(tU8 *serNum);

- tU8 isCard();
- tU8 readCardSerial();
- void init();

void writeMFRC522(tU8 addr, tU8 val):

Wysyła wartość (val) do rejestru o podanym adresie(addr).

1. Włączenie komunikacji z RC522

```
IOCLR = (1 << 7);
```

2. Wysłanie podanego adresu przez SPI.

```
SPI_Master_Write((addr << 1) & 0x7E);
```

3. Zapisanie wartości do wybranego rejestru.

```
SPI_Master_Write(val);
```

4. Wyłączenie komunikacji ze slave

```
IOSET = (1 << 7);
```

tU8 readMFRC522(tU8 addr):

Odczytuje wartość z podanego adresu.

1. Zmienna do przechowania odebranej wartości.

```
tU8 val;
```

2. Włączenie komunikacji z RC522

```
IOCLR = (1 << 7);
```

3. Wysyłanie adresu rejestru , z którego zostanie odczytana wartość

```
SPI_Master_Write(((addr << 1) & 0x7E) | 0x80);
```

4. Odczytanie wartości i jej zwrócenie.

```
val = SPI_Master_Read()
```

```
IOSET = (1 << 7);
```

```
return val;
```

void reset():

Soft reset czytnika.

1. Polega na wpisaniu do rejestru CommandReg wartości PCD_RESETPHASE (0x0F) , która odpowiada za SoftReset

```
writeMFRC522(CommandReg, PCD_RESETPHASE);
```

void setBitMask(tU8 reg, tU8 mask):

Ustawia wybrane bity podanego rejestru na 1.

1. Zmienna na starą wartość rejestru.

```
tU8 tmp;
```

2. Odczyt wartości rejestru.

```
tmp = readMFRC522(reg);
```

3. Ustawienie maski i wpisanie nowej wartości do rejestru.

```
writeMFRC522(reg, tmp | mask);
```

void clearBitMask(tU8 reg, tU8 mask):

Usuwa ustawioną wcześniej maskę.

1. Zmienna na starą wartość rejestru.

```
tU8 tmp;
```

2. Odczyt wartości rejestru.

```
tmp = readMFRC522(reg);
```

3. Usunięcie podanej maski.

```
writeMFRC522(reg, tmp & (~mask));
```

void init():

Inicjalizacja RC522

1. Wykonanie soft resetu.
`reset();`
2. Ustawienie timera do rejestru TModeReg wpisujemy wartość 0x8D(0000 0000 1000 1101) . Siódmy bit ustawiony na 1 oznacz, że timer będzie startował automatycznie pod koniec każdej transmisji niezależnie od sposobu komunikacji i prędkości. Jeśli w rejestrze RxModeReg bit RxMultiple nie jest ustawiony to taimer zatrzyma się automatycznie po odebraniu 5 bitów 1 startu i 4 danych. Cztery pierwsze bity definiują wartość TPrescaler.

```
writeMFRC522(TModeReg, 0x8D);
```

3. Definiujemy 8 dolnych bitów TPrescaler.(The following formula is used to calculate the timer frequency if the DemodReg register's TPrescalEven bit is set to logic 0: $f_{timer} = 13.56 \text{ MHz} / (2 * TPreScaler + 1)$. Where TPreScaler = [TPrescaler_Hi:TPrescaler_Lo] (TPrescaler value on 12 bits) (Default TPrescalEven bit is logic 0).

```
writeMFRC522(TPrescalerReg, 0x3E);
```

4. Ustawianie wartości TReloadReg (liczba , którą timer będzie dekrementował).

```
writeMFRC522(TReloadRegL, 30);
writeMFRC522(TReloadRegH, 0);
```

5. Wymuszenie modulacji 100% ASK (Amplitude Shift Keying). 100% amplitude modulation of RF carrier is used for communication from reader to card with modified miller encoding.

```
writeMFRC522(TxAutoReg, 0x40);
```

6. Sprawdzamy czy antena jest ustawiona poprawnie a jeśli nie to ją ustawiamy. Teoretycznie wymagane tylko wtedy gdy wystąpił reset przez RST.

```
antennaOn();
```

void antennaOn():

Sprawdza czy antena jest ustawiona poprawnie jeśli nie to jest ustawia.

1. Zmienna na wartość rejestru.

```
tU8 temp;
```

2. Odczyt wartości rejestru TxControlReg.

```
temp = readMFRC522(TxControlReg);
```

3. Jeśli 2 pierwsze bity nie są ustawione na jeden to je ustawia.

Jedynki na tych bitach oznaczają, że (output signal on pin TX2 TX11 delivers the 13.56 MHz energy carrier modulated by the transmission data).

```
if (!(temp & 0x03)) {
    setBitMask(TxControlReg, 0x03);
}
```

tU8 isCard():

Sprawdza czy karta została wykryta.

1. Zmienna do przechowania zwróconego statusu(karta została wykryta lub nie).

```
tU8 status;
```

2. Tablica szesnastu unsigned char na przechowywanie odczytanej wartości,

```
tU8 str[MAX_LEN];
```

3. Wysyłanie żądania (request) PICC_REQIDL – zmiana stanu karty na REDY

```
status = MFRC522Request(PICC_REQIDL, str);
```

4. Sprawdzenie czy karta została wykryta MI_OK == 0.

```
if (status == MI_OK) {
    return 1;
} else {
    return 0;
}
```

tU8 MFRC522Request(tU8 reqMode, tU8 *TagType):

Ustawia jak powinien wyglądać żądanie (request).

1. Zmienna na status operacji.

```
tU8 status;
```

2. Zmienna na ilość odczytanych bitów.

```
tU32 backBits;
```

3. Definiujemy ile bitów ostatniego bajtu ma być przesłanych.

```
writeMFRC522(BitFramingReg, 0x07);
```

4. Dane do wysłania.

```
TagType[0] = reqMode;
```

5. Wysyłanie do karty. PCD_TRANSCEIVE powoduje przygotowanie do wysłania danych z bufora FIFO do anteny i ustawienie automatycznego odbioru po wysłaniu wszystkich danych.

```
status = MFRC522ToCard(PCD_TRANSCEIVE, TagType, 1, TagType,
&backBits);
```

6. Jeśli przesłanie zakończyło się poprawnie to zwraca odpowiedni status.

```
if ((status != MI_OK) || (backBits != 0x10))
{
    status = MI_ERR;
}
```

```
return status;
```

tU8 MFRC522ToCard(tU8 command, tU8 *sendData, tU8 sendLen, tU8 *backData, tU32 *backLen):

Wysyła i odczytuje dane z karty.

tU8 command – komenda wysyłana do rejestru CommandReg

tU8 *sendData – dane wysyłane do bufora FIFO i karty

tU8 sendLen – długość wysyłanych danych (ile bajtów)

tU8 *backData – odczytane dane z bufora FIFO

tU32 *backLen – długość odczytanych danych (ile bajtów)

1. Zmienne startowe

- tU8 status = MI_ERR; - MI_ERR = 2 status start niepowodzenie
- tU8 irqEn = 0x77; - które przerwania mają być włączone
- tU8 waitIRq = 0x30; - które przerwania powinny wystąpić
- tU8 lastBits; - pomocnicza zmienna
- tU8 n; - zmienna pomocnicza
- tU32 i; - iterator do while

2. Włączenie wszystkich przerwań.

0x77 | 0x80 (1111 0111)

0 TimerIRq allows the timer interrupt request (TimerIRq bit) to be propagated to pin IRQ

1 ErrIRq allows the error interrupt request (ErrIRq bit) to be propagated to pin IRQ

2 LoAlertIRq allows the low alert interrupt request (LoAlertIRq bit) to be propagated to pin IRQ

4 IdleIRq allows the idle interrupt request (IdleIRq bit) to be propagated to pin IRQ

5 RxIRq allows the receiver interrupt request (RxIRq bit) to be propagated to pin IRQ

6 allows the transmitter interrupt request (TxIRq bit) to be propagated to pin IRQ

7 if 1 signal on pin IRQ is inverted with respect to the Status1Reg register's IRq bit

```
writeMFR522(CommIRqReg, irqEn | 0x80);
```

3. Ustawienie na 0 siódmego bitu rejestru CommIRqReg powoduje, że bity odpowiedzialne za informacje o tym czy dane przerwanie wystąpiło zostały wyzerowane.

```
clearBitMask(CommIRqReg, 0x80);
```

4. Ustawienie na 1 siódmego bitu rejestru CommIRqReg powoduje, że bity mogą zostać ustawione na 1 gdy wystąpi dane przerwanie.

```
setBitMask(FIFOLevelReg, 0x80);
```

5. Zatrzymanie aktualnie wykonywanych komend.

PCD_IDLE – 0x00 no action, cancels current command execution

```
writeMFR522(CommandReg, PCD_IDLE);
```

6. Wysyłanie danych do bufora FIFO

```
for (i = 0; i < sendLen; i++) {
    writeMFR522(FIFODataReg, sendData[i]);
}
```

7. Wysłanie command do rejestru CommIrqReg w tym miejscu w naszym program będzie to zawsze PCD_TRANSCEIVE = 0x0C. PCD_TRANSCEIVE powoduje przygotowanie do wysłania danych z bufora FIFO do anteny i ustawienie automatycznego odbioru po wysłaniu wszystkich danych.

```
writeMFR522(CommandReg, command);
```

8. Ustawienie siódmego bitu w rejestrze BitFramingReg powoduje rozpoczęcie wysyłania danych z bufora FIFO.

```
setBitMask(BitFramingReg, 0x80);
```

9. Oczekiwanie na wystąpienie odpowiednich przerwań.
CommIrqReg – informuje, które przerwania wystąpiły
0x01 – timer zdekrementował swoją wartość do zera
waitIRq(0x30) – bit 5 na 1 oznacza, że dane zostały odebrane z poprawnym zakończeniem
bit 4 ustawiony na 1 oznacza, że komenda wysyłania zakończyła się.

```
i = 2000;
do {
    n = readMFR522(CommIrqReg);
    i --;
} while ((i != 0) && !(n & 0x01) && !(n & waitIRq));
```

10. Wyłączenie możliwości wysyłania z FIFO.

```
clearBitMask(BitFramingReg, 0x80);
```

11. Reszta kodu wykonywana jest jeśli i != 0 co oznacza że do czasu dekrementacji wartości timera została odebrana odpowiedź od karty.

12. Sprawdzenie rejestru błędów ErrorReg jeśli żaden z nich nie wystąpił program jest kontynuowany w przeciwnym razie status = MI_ERR.

```
(0x1B) 0001 1011
```

0 - ProtocolErr 1set to logic 1 if the SOF is incorrect

1 - ParityErr1parity check failed

3 - CollErr1a bit-collision is detected

4 – BufferOvfl

13. Ponowne sprawdzenie czy wszystko zostało wykonane poprawnie.

```
if (n & irqEn & 0x01) {
    status = MI_NOTAGERR; // MI_NOTAGERR = 1;
}
```

14. Odczyt ile bitów znajduje się w kolejce FIFO.

```
n = readMFRC522(FIFOLevelReg);
```

15. Odczytanie ilości poprawnie odczytanych bitów z ostatnio przeczytane go byte.

0x07 – liczba tych bitów zapisan jest na 3 pierwszych bitach

```
lastBits = readMFRC522(ControlReg) & 0x07;
```

16. Jeśli jakiegokolwiek bity są valid to zapisujemy do *backLen długość w bitach , jeśli odczytano więcej niż 16 byte to ustawiamy n na 16 , odczytujemy i zapisujemy w *backData odpowiednią ilość odczytanych bytów.

tU8 anticoll(tU8 *serNum):

1. Zmienne startowe

tU8 status - status operacji

tU8 i - iterator do pętli for

tU8 serNumCheck – zmienna pomocnicza do sprawdzenia czy nie wystąpiła kolizja

tU32 unLen – ile bitów odczytano

2. Wyczyszczenie BitFramingReg w wypadku gdyby zostały jakieś pozostałości po wykonanych wcześniej operacjach.

```
writeMFRC522(BitFramingReg, 0x00);
```

3. Zapisujemy do serNum dane do wysłania.

```
serNum[0] = PICC_ANTICOLL // uruchomienie antykolizyjnej pętli
serNum[1] = 0x20 - randomowy numer i tak nie zostanie przesłane ze
względu na 3 pierwsze bity BitFramingReg
```

4. Przesłanie danych do karty jeśli zakończyło się powodzeniem to status == MI_OK.

```
status = MFRC522ToCard(PCD_TRANSCEIVE, serNum, 2, serNum, &unLen);
```


5. Sprawdzenie czy wystąpiła kolizja. Jeśli kolizja nie wystąpiła to XOR czterech pierwszych bitów powinien być równy piątemu.

```

if (status == MI_OK)
{
    for (i = 0; i < 4; i++)
    {
        serNumCheck ^= serNum[i];
    }
    if (serNumCheck != serNum[i])
    {
        status = MI_ERR;
    }
}

return status;

```

tU8 readCardSerial():

Odczytuje numer seryjny karty ma on 16 bajtów a pierwsze 4 to ID.

1. Zmienne startowe:

```

tU8 status - status operacji
tU8 str[MAX_LEN] - odczytane wartości z bufora FIFO

```

2. Włączenie pętli antykolizyjnej i odczytanie ID karty.

```
status = anticoll(str);
```

3. Kopiowanie odczytanego id do serNum. Jeśli nie wystąpiła kolizja funkcja zwraca 1.

```

static tU8 i;
for (i = 0; i < 4; i++) {
    *(serNum + i) = *(str + i);
}

if (status == MI_OK) {
    return 1;
} else {
    return 0;
}

```

tU8 checkIfValidTag():

Sprawdza czy odczytany tag jest zgodny z tym zakodowanym w programie.

1. Tag karty , która może otworzyć zamek.

```
tU8 validTag[4] = {0xe7, 0x92, 0xde, 0x03};
```

2. Sprawdzenie czy tag zgadza się z tym odczytanym z przyłożonej karty. Jeśli nie zwraca FALSE.

```
for (i = 0; i < 4; i++) {  
    if(serNum[i] != validTag[i]) {  
        return FALSE;  
    }  
}
```

3. Czyszczenie pamięci gdzie został zapisany odczytany tag.

```
for (i = 0; i < 4; i++){  
    serNum[i] = 0x00;  
}
```

4. Zwraca TRUE po wyczyszczeniu tablicy serNum.

```
return TRUE;
```

3.3 TIMER

```
void udelay(tU32 delayInUs) {
    T1TCR = 0x02;
    T1PR = 0x00;
    T1MR0 = (((long)delayInUs-1) * (long)CORE_FREQ/1000) / 1000;
    T1IR = 0xff;
    T1MCR = 0x04;
    T1TCR = 0x01;
    while (T1TCR & 0x01)
        ;
}
```

3.3.1 Rejestry

Wykorzystane rejestry w programie:

- **T1TCR – Time Control Register**, używany do kontroli pracy timera/countera.
- **T1PR – Prescale Register**, określa maksymalną wartość dla **Prescale Counter**.
- **T1MR0 – Match Register**, jest porównywany z wartością **Timer Counter**, jeśli obydwie wartości są identyczne, jest wykonywana akcja określona przez rejestr **MCR**.
- **T1IR – Interrupt Register**, rejestr służący do pracy z przerwaniami.
- **MCR – Match Control Register**, służy do zdefiniowania operacji jaka zostanie wykonana jeśli **Match Register = Timer Counter**.

Wartości poszczególnych rejestrów:

- **T1TCR**, Ustawienie bitu 0 na wartość 1 powoduje uaktywnienie Timer Counter i Prescale Counter; ustawienie bitu 1 na wartość 1 powoduje zresetowanie Timer Counter i Prescale Counter.

$$T1TCR = 0x02;$$
- **T1PR**, Maksymalna wartość countera wynosi 0.

$$T1PR = 0x00;$$
- **T1MR0**, ilość inkrementacji **TC** potrzebna do osiągnięcia zadanego czasu w mikrosekundach $(10e-6 * 10e6 / 1000) / 1000 = (10e-6 * 10e6 * 10e-3) * 10e-3 = 10e-3 * 10e-3 = 10e-6$ [CORE_FREQ taktowanie w MHz, delayInUs żądany czas w uS]. Wynik 10e-6, tj. 10e-6s, czyli uzyskany czas to faktycznie mikro sekundy (10e-6 = mikro).

$$T1MR0 = (((long)delayInUs-1) * (long)CORE_FREQ/1000) / 1000;$$
- **T1IR**, Rejestr ośmiobitowy, wpisanie jedynki na dany bit powoduje zresetowanie przerwania, wpisanie 0 nie wywołuje żadnej efektu. Wartość 0xff powoduje reset wszystkich przerw.

```
T1IR = 0xff;
```

- **T1MCR**, ustawienie 2 bitu na 1 powoduje zatrzymanie **TC** i **PC** oraz ustawienie wartości 0 bitu rejestru **TCR** na 0 jeśli **TC** osiągnie wartość **MR0**.

```
T1MCR = 0x04;
```
- **T1TCR**, ustawienie 1 bitu na 0 powoduje wyprowadzenie timera ze stanu resetu.

3.3.2 Działanie kodu

Funkcja **void udelay** posiada jeden parametr, zmienną typu **tU32** – zadany czas podany w mikrosekundach. Samo działanie jest proste, program pozostaje w nieskończonej pętli dopóki na bicie 0 rejestru **TCR** nie zostanie ustawiona wartość 1, co będzie równoznaczne z osiągnięciem wartości rejestru **MR0** przez rejestr **TC**.

3.4 GPIO (Dioda oraz guziki)

W programie obsługujemy diodę oraz trzy guziki. Guziki znajdują się na pinach P1.20, P1.21, P1.22 oraz P1.23. Dioda jest podłączona zewnętrznie i obsługujemy ją za pomocą pinu P0.10. Aby skorzystać z diody oraz guzików należy skonfigurować rejestry PINSEL oraz PINDIR w następujący sposób:

- Ustawić bity w rejestrach PINSEL1 oraz PINSEL0 odpowiadające za piny, które obsługujemy (na wartość 00) w celu uzyskania trybu GPIO

```
PINSEL0 |= (0 << 21) | (0 << 20);
```
- Ustawić bity w rejestrze IODIR1 oraz IODIR0 w celu wskazania kierunku, w naszym przypadku P0.10 na wyjście, a P1.20-23 na wejście

```
IODIR0 |= (1 << 10);
```
- Kontrolowanie diody odbywa się poprzez ustawienie bitu na pozycji 10 za pomocą IOSET0 (włączenie diody, ustawienie stanu wysokiego) oraz IOCLR0 (wyłączenie diody, ustawienie stanu niskiego). W przypadku guzików odczytujemy z rejestru IOPIN1 aktualny stan pinów, następnie wykonujemy operacje AND z wartością przycisku, który oczekujemy otrzymać.

```
IOSET0 = (1 << 10); //włączenie diody
IOCLR0 = (1 << 10); //wyłączenie diody
(IOPIN1 & key) == 0;
```

3.5 Obsługa przycisków z tłumieniem drgań

Zaproponowane rozwiązanie eliminujące tzw. Bounce jest rozwiązaniem programowym zrealizowanym przez poniższy kod.

3.5.1 Kod

```
tU8 checkButton(tU32 key) {
    tU8 wcisnietyKlawisz;
    if ((IOPIN1 & key) == 0) {
        osSleep(5);
        if ((IOPIN1 & key) == 0) {
```

```

        wcisnietyKlawisz = key;
        return TRUE;

    } else {
        wcisnietyKlawisz = 0;
        return FALSE;
    }
} else {
    wcisnietyKlawisz = 0;
    return FALSE;
}
}

```

3.5.1 Działanie mechanizmu tłumienia drgań

Działanie polega na sprawdzeniu wciśnięcia klawisza, następnie odczekania 50ms za pomocą funkcji **osSleep(5)** i ponownemu sprawdzeniu klawisza, jeśli dalej jest wciśnięty funkcja zwróci wartość **TRUE**, w każdym przeciwnym wypadku – **FALSE**.

3.6 PWM + Buzzer

Dźwięk o danej częstotliwości jest generowany przez PWM. W naszym programie wykorzystujemy pin P0.9 przełączony w tryb PWM. Inicjalizacja PWM wygląda następująco:

```

void initPWM(tU8 frequency) {
    PINSEL0 |= 0x00080000;
    PWM_PR = 0x00+60;
    PWM_MCR = 0x02;
    PWM_MR0 = frequency;
    PWM_MR6 = PWM_MR0 / 2;
    PWM_LER = 0x41;
    PWM_PCR = 0x4000;
    PWM_TCR = 0x00;
}

```

- Ustawienie pinu P0.9 na tryb PWM6.

```
PINSEL0 |= 0x00080000;
```

- Ustawienie prescaler'a na 60. Wartość rejestru PWM_TC jest inkrementowana po upływie ilości cykli zawartych w rejestrze PWM_PR + 1.

```
PWM_PR = 0x00+60;
```

- W przypadku gdy PWM_TC będzie miał taką samą wartość co rejestr PWM_MR0 nastąpi wyzerowanie rejestru PWM_TC.

PWM_MCR = 0x02;

- Ustawienie okresu impulsu.

PWM_MR0 = frequency;

- Współczynnik wypełnienia, określa jaką część okresu stanowi stan wysoki (logiczna jedynka).

PWM_MR6 = PWM_MR0 / 2;

- Ustawienie to określa, że po upływie całego okresu podczas resetu rejestru PWM_TC wartości rejestrów PWM_MR0 oraz PWM_MR6 otrzymają ostatnio wpisane wartości.

PWM_LER = 0x41;

- Ustawienie PWM6 na wyjście oraz PWM6 na „single edge”.

PWM_PCR = 0x4000; ZMIENIĆ W PROGERAMIE NA TĄ LICZBE!!!!

- Początkowo ustawiamy na 0x00: licznik oraz PWM wyłączony.

PWM_TCR = 0x00;

Do obsługi PWM'a wykorzystujemy dwie funkcje: pwmON() oraz pwmOFF(). W funkcji pwmON() wprowadzamy do rejestru PWM_TCR wartość 0x09 w celu uruchomienia licznika oraz PWM'a:

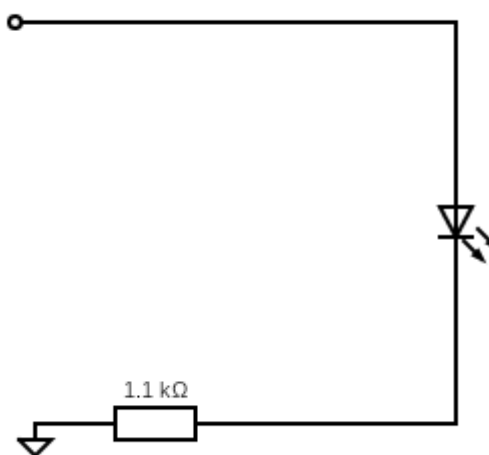
PWM_TCR = 0x09;

Analogicznie w przypadku funkcji pwmOFF() w której również wprowadzamy nową wartość do rejestru PWM_TCR tym razem o wartości 0x02 w celu wyłączenia licznika, PWM'a oraz zresetowaniu wartości rejestrów PWM_TC oraz PWM_PC:

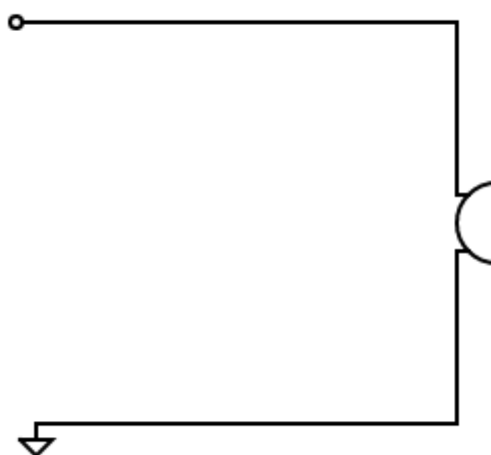
PWM_TCR = 0x02;

Do pinu P0.9 jest podpięty bezpośrednio brzęczyk który generuje dźwięk o zadanej częstotliwości. Za pomocą funkcji pwmON() oraz pwmOFF() jesteśmy w stanie kontrolować kiedy brzęczyk ma generować dźwięk.

3.7 Schemat układu elektronicznego



Rys. 1 – przedstawia obwód z diodą LED.



Rys. 2 – przedstawia obwód z bucikiem.

Dioda LED obsługiwana jest za pomocą pinu P0.10 ustawionego w tryb pracy GPIO, zaś buczonek sterowany jest za pomocą PWM'a.

4. Analiza skutków awarii

Poniżej opisano wpływ skutków awarii poszczególnych podzespołów na działanie systemu. Poprzez awarię podzespołu należy rozumieć uszkodzenie samego podzespołu, jak i uszkodzenie elementów łączących dany podzespół z mikrokontrolerem.

Skutki awarii czytnik RFID i przycisków - Aby otworzyć zamek należy do czytnika przyłożyć kartę, której ID zostało zapisane w systemie. W przypadku jego awarii istnieje dodatkowa możliwość otwarcia zamka przy użyciu przycisków, które muszą zostać wciśnięte w odpowiedniej kolejności. Awarię czytnika lub jednego z przycisków można traktować jako średnie zagrożenie dla systemu ponieważ ciągle istnieje możliwość otwarcia zamka. W przypadku awarii obu podzespołów należy traktować to jako awarię krytyczną.

Wykrywanie awarii czytnika RFID i przycisków – w czytniku RFID wiele elementów może ulec uszkodzeniu dlatego w naszym systemie nie jest sprawdzane, który konkretnie element zawiódł. Najprostszym sposobem wykrycia czy system jest w stanie komunikować się z czytnikiem jest wysłanie rozkazu który zwróci wersję systemu, jeśli jest ona poprawna to wiadomo, że komunikacja między systemem a czytnikiem działa. Awaria czytnika może pojawić się też w części odpowiedzialnej za wysyłanie rozkazów do karty mifare np. awaria anteny oraz awarii może ulec też sama karta jednak nie jest to wykrywane. Przyciski traktowane są jako awaryjny sposób otwierania zamka jedyny sposób na sprawdzenie czy działają poprawnie jest próba otworzenia zamka za ich pomocą.

Skutki awarii diody – Dioda w naszym systemie reprezentuje zamek dlatego jej awaria jest krytyczna dla systemu ponieważ uniemożliwia otwarcie zamka.

Wykrywanie awarii diody – Nasz system nie wykrywa czy dioda uległa awarii. Jedyny sposób na stwierdzenie, że to rzeczywiście dioda przestała działać jest próba otwarcia zamka. Jeśli przy jego otwieraniu dioda nie świeci się ale buczek wydaje ton świadczący to o awarii diody lub pinu do którego jest podpięta.

Skutki awarii buzzera – Buzzer nie ma żadnego wpływu na działanie programu gdyż pewni on tylko funkcję informacyjną że zamek jest otwarty.

Wykrywanie awarii buzzera – Awaria ta nie jest wykrywana. Można stwierdzić czy buzzer uległ uszkodzeniu w taki sam sposób jak można sprawdzić diodę.

Skutki awarii mikrokontrolera – w mikrokontrolerze awaria może wystąpić w wielu miejscach i nie wpłynąć na działanie systemu np.: awaria nieużywanych pinów, jednak może wystąpić taka awaria, że mikrokontroler przestanie być zdatny do użytku.

Wykrywanie awarii mikrokontrolera – w naszym systemie nie jest to wykrywane.

Nr.	Awaria podzespołu	Skutki awarii
1.	czytnik RFID	Średnie
2.	dioda	Krytyczne
3.	buzzer	Niegroźne
4.	przyciski	Średnie
5.	mikrokontrolera	Krytyczne

Poszczególne skutki należy rozumieć następująco:

Niegroźne - awaria podzespołu ogranicza funkcjonalność systemu, jednak nie wpływa na poprawność jego działania.

Średnie - awaria pojedynczego podzespołu powinna być traktowana jako awaria niegroźna jednak w przypadku awarii wielu podzespołów oznaczonych jako średnie należy traktować je jako awarie krytyczne.

Krytyczne - awaria podzespołu może doprowadzić do niemożności wykonywania się programu lub ograniczenia funkcjonalności na tyle, że system nie może wykonywać zadania do, którego został stworzony.

5. Źródła

- <https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf>
- <https://github.com/miguelbalboa/rfid>
- https://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf
- <https://www.nxp.com/docs/en/user-guide/UM10120.pdf>