KARKONOSKA PAŃSTWOWA SZKOŁA WYŻSZA
w Jeleniej Górze
WYDZIAŁ PRZYRODNICZO-TECHNICZNY

PRACA DYPLOMOWA

Wykorzystanie serwomechanizmu do zabezpieczenia drzwi przed niepożądanym otwarciem

Grzegorz Czernatowicz

Praca napisana pod kierunkiem:

dr inż. Jerzy Pietruszewski

Jelenia Góra, 2016

Spis treści:

1. Wstęp

Szybki rozwój technologi cyfrowych oraz opracowywanie coraz bardziej wydajnych metod produkcji elementów elektronicznych i układów scalonych powoduje ciągły spadek cen i wzrost popularności elektroniki konsumenckiej. Wolne i otwarte oprogramowanie zyskuje coraz większe rzesze zwolenników i dotyczy to również elektroniki. Wiele komercyjnych technologii i patentów ma swoje darmowe odpowiedniki, z których może korzystać każdy, pod warunkiem posiadania odpowiedniej wiedzy.

Oprogramowanie i układy elektroniczne na licencji GPL (ang. *General Public License*), zyskują coraz większy udział na rynku elektroniki konsumenckiej. Znane projekty takie jak Linux, OpenOffice, czy Firefox oparte są na licencji wolnego i otwartego oprogramowania.

Jednym z takich projektów jest Arduino (rys. 1). Jest to projekt, który łączy platformę programistyczną opartą na otwartym i wolnym oprogramowaniu, gotowe obwody drukowane z mikrokontrolerem oraz społeczność tworzącą oprogramowanie i urządzenia oparte na tym projekcie. Projekt bazuje na modelach produkowanych przez kilku sprzedawców wykorzystujących różne mikrokontrolery. Są to kompletne systemy, które zapewniają zestaw cyfrowych i analogowych gniazd, do których można podłączać dowolne moduły lub układy rozszerzające funkcjonalność. Systemy te programuje się wykorzystując środowisko programistyczne Arduino IDE, które oparte jest na językach programowania Wiring oraz C/C++. Arduino pojawiło się pierwszy raz w 2005 roku we Włoszech jako projekt jednego ze studentów uczelni w mieście Ivrea. Nazwa pochodzi od baru, w którym spotykali się tamtejsi studenci. Celem Arduino jest dostarczenie tanich i łatwych w programowaniu urządzeń, które mogą być wykorzystywane do projektów nawet przez osoby, które nie miały wcześniej styczności z elektroniką.

Rysunek 1: Płytka Arduino, model Uno R3

Rozbudowana dokumentacja i łatwy dostęp do tysięcy projektów utworzonych przez społeczność, uczyniły Arduino jedną z najpopularniejszych platform do tworzenia sprzętu zdolnego do interakcji z analogowym, rzeczywistym światem. Prostota budowania prototypów pozwala na eksperymentowanie z różnymi modułami, a przemyślana konstrukcja umożliwia wielokrotne wykorzystanie płytki w różnych projektach. Pozytywny wpływ na popularność platformy ma również duży rynek klonów modeli Arduino. Urządzenia te bazują na tańszych zamiennikach oryginalnych części zapewniając jednak pełną kompatybilność z Arduino IDE. Oprogramowanie i sprzęt Arduino jest oparty o licencję open source, która zachęca producentów do tworzenia lepszych lub tańszych wersji. Najtańszy klon Arduino Uno R3 można kupić już od ośmiu złotych.

Razem z rozwojem Arduino powstało pojęcie przetwarzania namacalnego(ang. *physical computing*). Przetwarzanie namacalne jest rozszerzeniem przetwarzania bez granic. Polega ono na wykorzystaniu urządzeń elektronicznych w każdej dziedzinie życia w taki sposób, by zwykły użytkownik nie wiedział o obecności mikrokontrolera w danym przedmiocie. Jest to również tworzenie interaktywnych układów fizycznych wykorzystując oprogramowanie i urządzenia zdolne do przetwarzania danych z czujników analogowych.

Praktycznie, pojęcie to opisuje wykonane ręcznie dzieła artystyczne, które wykorzystują różnego rodzaju czujniki, jak i również sterowniki do kontroli urządzeń

elektro-mechanicznych takich jak silniki elektryczne, serwomechanizmy czy oświetlenie.
Arduino rozwijane jest z myślą o projektantach i artystach – osobach o niewielkiej wiedzy technicznej. Arduino umożliwia budowanie wyrafinowanych projektów prototypów i interaktywnych instalacji artystycznych osobom nawet bez doświadczenia w programowaniu. Poradniki wyjaśniające każdy krok pozwolą szybko zbudować każdemu swój pierwszy układ. Przyswojenie podstaw programowania tych układów jest proste, ale jednocześnie pomaga osobom bardziej zaawansowanym technicznie w szybszym tworzeniu swoich projektów, bez konieczności budowania całego układu od podstaw i bez żadnych specjalistycznych narzędzi do lutowania elementów elektronicznych. Istnieje wiele oficjalnych modeli i ich kompatybilnych klonów. Oficjalne modele to m.in. Uno, Pro, Mini, Micro i Nano. Ciekawy jest model Lilypad. Jest on przystosowany do mocowania na ubiorze, by umożliwić tworzenie interaktywnej odzieży.

2. Cel i zakres pracy

Celem pracy było stworzenie urządzenia do zabezpieczenia drzwi przed niepożądanym otwarciem, wykorzystując serwomechanizm oraz inne moduły sterowane przez mikrokontroler. Niniejsza praca może zostać wykorzystana przez przyszłych studentów do rozbudowania możliwości przedstawionego urządzenia. Przykładowym tematem takiej pracy mogłoby być stworzenie systemu zamków centralnie sterowanych przez główny serwer, oparty np. na miniaturowym komputerze Raspberry Pi.

Wybór tego tematu wiąże się z chęcią stworzenia taniej wersji zamka elektronicznego i zastąpienie drogich zamków przemysłowych. Ukończone urządzenie byłoby specjalną nakładką montowaną na istniejące już drzwi z zamkiem mechanicznym, bez konieczności (lub tylko niewielkiej, np. nawiercenie niewielkich otworów) ingerencji w ich konstrukcję. Urządzenie to ma też wyróżniać się niskim kosztem budowy. Zastosowane elementy i moduły muszą być jak najtańsze, by utrzymać koszt budowy i koszty ewentualnej naprawy na możliwie jak najniższym poziomie.

Zakres pracy obejmuje następujące zagadnienia:

- przegląd literatury na temat rodzajów magistral i metod wykorzystywanych do komunikacji mikrokontrolera z innymi urządzeniami,
- przegląd dokumentacji technicznej poszczególnych podzespołów, opisanie zasad ich działania,
- przybliżenie funkcjonalności oprogramowania służącego do programowania mikrokontrolerów,
- zaprojektowanie układu elektronicznego zawierającego wcześniej wymienione elementy,
- stworzenie programu obsługującego urządzenia we wcześniej stworzonym układzie elektronicznym,
- zbudowanie prototypu urządzenia wraz z makietą w postaci miniaturowych drzwi,
- rozważania na temat możliwości rozszerzenia funkcjonalności urządzenia,

3. Technologie wykorzystywane w mikrokontrolerach

W rozdziale tym będzie przedstawiony rys historyczny, opisujący w skrócie wydarzenia, które w dużym stopniu przyczyniły się do rozwoju technologii. Zaprezentowane zostaną również technologie wykorzystywane w nowoczesnych systemach wbudowanych, m.in. metody transmisji danych pomiędzy różnymi podzespołami.

3.1 Rys historyczny

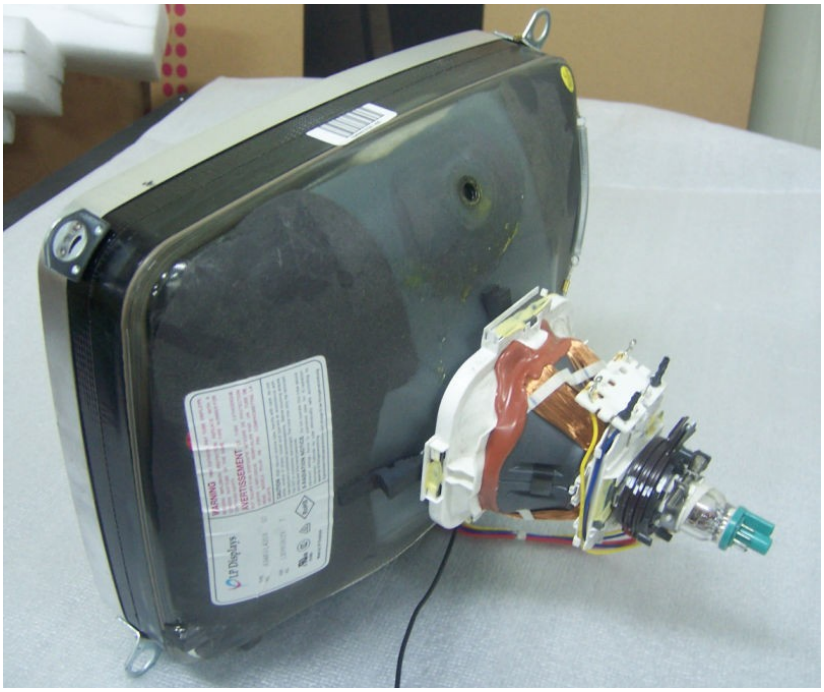
Początki elektroniki sięgają XIX wieku, w którym powstały fundamenty współczesnej elektrotechniki, m. in. prawo Ohma (1826 rok) i pierwsze prawo Kirchhoffa, sformułowane w 1845 roku. Na początku XX wieku John Ambrose Fleming opracował pierwszą diodę próżniową (rys. 2)[1].

Rysunek 2: Lampa elektronowa

1 [7] Wójcik Marcin (2000), hierarchizacja ważności wynalazków wpływających na postęp elektroniki, Akademia Górniczo – Hutnicza, Kraków, rozdział 4

W latach dwudziestych XX wieku rozpoczął się szybki rozwój elektroniki – powstają pierwsze komercyjne stacje radiowe. Pierwsza wojna światowa przyśpieszyła rozwój technik radiowych, by ulepszyć możliwości komunikacyjne wojska.

Pierwszy kineskop został wynaleziony w Rosji w 1929 roku przez Vladimir Zworykin. Wykorzystał on lampę elektronową wyposażoną w ekran do wyświetlenia sygnału video, który zawierał proste kształty geometryczne, stając się pionierem znanej nam dziś telewizji. Tak powstał pierwszy kineskop (rys. 3). W tym samym czasie, w roku 1924, odbyła się pierwsza transmisja telewizyjna z Londynu do Nowego Jorku [2].



Rysunek 3: Kineskop

W 1926 powstała lampa próżniowa Loewe 3NF, która uważana jest za pierwszy układ scalony. Zawierała w sobie kilka elementów takich jak triody, rezystory i kondensatory. Po podłączeniu baterii, cewki i głośnika można było otrzymać kompletny odbiornik radiowy.

W 1947 roku zbudowany został pierwszy ostrzowy tranzystor, którego wynalezienie spowodowało bardzo szybki rozwój elektroniki. Układy elektroniczne były stosowane w coraz większej liczbie gałęzi przemysłowych czy w życiu codziennym.

Pierwszy, w pełni automatyczny, programowalny komputer został zbudowany przez

2 Ibidem, rozdział 4.2

niemieckiego inżyniera Konrada Zuse w 1941 roku[3]. Miał on zaimplementowane wykonywanie instrukcji w pętli, jednak nie posiadał możliwości wykonania instrukcji warunkowych.

Mimo to, komputer Z3 jest kompletną maszyną w sensie Turinga – można go wykorzystać do wykonania dowolnego algorytmu[4].

Ferranti Mark 1 był pierwszym dostępnym komercyjnie elektronicznym komputerem, który dostępny był od roku 1951 (rys. 4). Na komputerze stworzone zostało pierwsze nagranie muzyki cyfrowej, jak i również jedna z najstarszych gier komputerowych.

Rysunek 4:Ferranti Mark 1

Wynalezienie tranzystora pozwoliło na zastąpienie dużych, prądożernych lamp próżniowych mniejszymi elementami. W roku 1958 Jack Kilby przedstawił prototyp działającego miniaturowego układu scalonego[5]. Pierwsze użycie terminu „mikroprocesor" przypisywane jest firmie Viatron, opisujące układ scalony wykorzystany w komputerze System 21, którego sprzedaż rozpoczęła się w roku 1968. Intel 4004 był pierwszym 4-bitowym procesorem, który ukazał się w roku 1971. Przetwarzał 4-bitowe dane, ale

3 Ibidem, rozdział 4.2

4 "How to Make Zuse's Z3 a Universal Computer", [8] http://www.inf.fu-berlin.de/users/rojas/1997/Universal_Computer.pdf [dostęp 11.02.2016]

5 The Chip that Jack Built, [15] http://www.ti.com/corp/docs/kilbyctr/jackbuilt.shtml [dostęp 28.01.2016]

wykorzystywał do tego czterdzieści sześć 8-bitowych instrukcji. Częstotliwość zegara wynosiła 740 kHz. W roku 1974 został stworzony pierwszy mikrokontroler TMS 1000 (rys 5)[6].

Rysunek 5:Mikrokontroler TMS1000

3.2 Mikrokontroler, a mikroprocesor

Mikrokontroler i mikroprocesor to dwa różne układy scalone. Jednak można je ze sobą pomylić, gdyż oba wykorzystywane są do podobnych zadań i niektóre mają identyczną zewnętrzną obudowę. Różnice techniczne pomiędzy nimi są jednak bardzo duże. Mikroprocesor posiada kilka elementów[7]:

- jednostka arytmetyczno-logiczna – wykonuje proste operacje arytmetyczne i logiczne na liczbach całkowitych,
- rejestry – przechowują dane oraz wyniki działań,
- logiczna jednostka sterująca – przekazuje instrukcje z pamięci i kontroluje wykonywanie programu,

Zastosowanie mikroprocesora nie jest z góry określone. Może być on wykorzystywany w dowolnym urządzeniu przetwarzającym duże ilości danych. Mikroprocesor służy do uruchamiania dużych, rozbudowanych programów.

Mikrokontroler, w przeciwieństwie do mikroprocesora, zawiera w sobie kilka dodatkowych elementów, co pozwala zmniejszyć liczbę styków potrzebnych do jego podłączenia. Większość mikrokontrolerów składa się z:

- jednostki arytmetyczno-logicznej
- pamięci RAM (ang. *Random Access Memory*) – do przechowywania danych,

6 Intel and TI – Microprocessors and Microcontrollers, [10] http://www.datamath.org/Story/Intel.htm [dostęp 08.02.2016]

7 [2] Deshmukh V Ajay (2005), Microcontrollers: Theory and Applications, Tata McGraw-Hill Education, rodział 1

- pamięci ROM (ang. *Read Only Memory*) – przechowuje kod programu, najczęściej jest to pamięć typu Flash,
- timerów,
- układu watchdog – restartuje mikrokontroler w przypadku nieprawidłowego działania,
- szeregowych portów wejścia/wyjścia,
- uniwersalnych portów wejścia/wyjścia,
- kontrolery przerwań,
- kontrolery transmisji szeregowej,
- obszar pamięci nieulotnej

Mikrokontroler jest więc kompletnym komputerem zawartym na jednym obwodzie drukowanym. Można kupić gotowe obwody drukowane zawierające mikrokontroler oraz wszystkie potrzebne elementy np. gniazda dla złącz wejścia/wyjścia lub przygotować taki układ samodzielnie.

Mikrokontrolery wykorzystywane są w systemach wbudowanych przystosowanych do pracy autonomicznej, czyli automatycznego sterowania innymi urządzeniami. Najczęściej stosowane są w urządzeniach RTV i AGD, robotach przemysłowych czy zabawkach. Koszt ich produkcji jest dużo niższy, niż mikroprocesorów.

3.3 Modulacja szerokości impulsów (PWM)

Fundametalnym modułem mikrokontrolera jest *Timer* (ang. *licznik*). Każdy *Timer* posiada kilka trybów pracy. Jednym z nich jest PWM (ang. *Pulse-width modulation*)

Sześć pinów cyfrowych obsługuje PWM. Modulacja szerokości impulsu jest metodą, która pozwala na wygenerowanie zmiennego napięcia używając sygnału cyfrowego. Zmienne napięcie jest cechą sygnału analogowego, a układy cyfrowe potrafią wygenerować tylko dwa stany, wysoki lub niski. Sygnał analogowy może przyjmować dowolne napięcie z przedziału w dowolnym czasie. Jeżeli będziemy zmieniać sygnał cyfrowy z niskiego na wysoki w spójny sposób, dostaniemy stałą proporcję pomiędzy napięciem logicznym wysokim, a niskim. Przy napięciu 5V, jeżeli stan wysoki utrzymamy przez 1 mikrosekundę, a następnie stan niski również przez jedną mikrosekundę, otrzymamy napięcie 2.5 V. Zakłócenia wynikające z częstego przełączania można zniwelować wykorzystując cewkę i kondensator.

Najważniejszym parametrem sygnału PWM jest współczynnik wypełnienia[8]:

$$k_w = \frac{T_{ON}}{T}$$

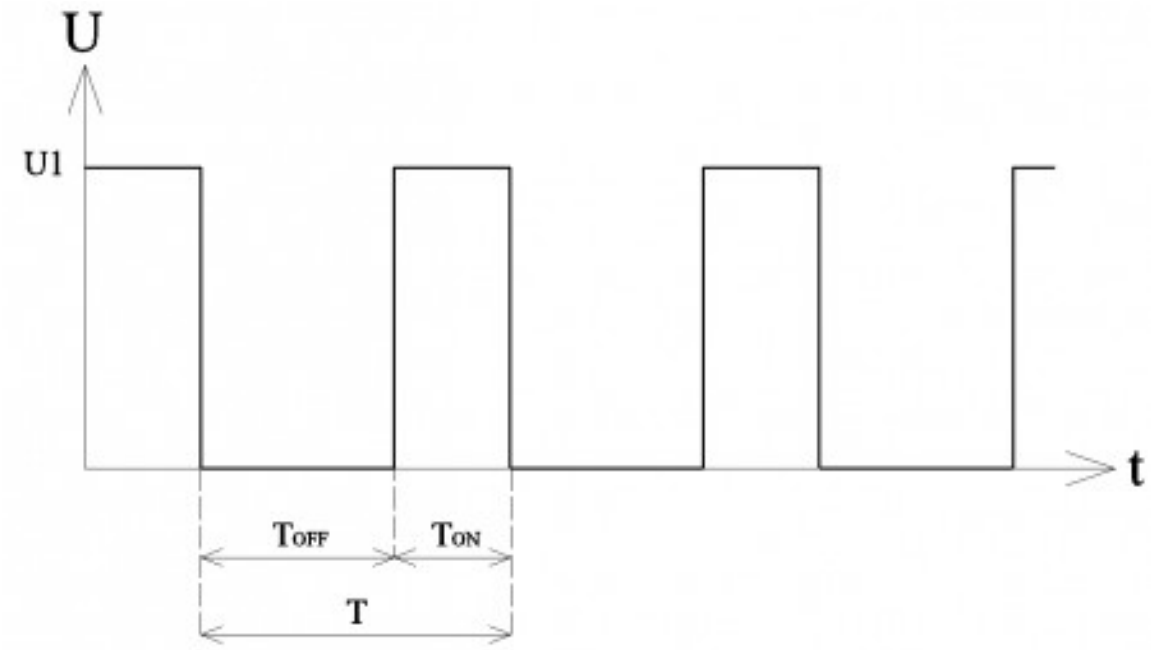
gdzie:

$T = T_{ON} + T_{OFF}$

T_{ON} = czas stanu wysokiego sygnału PWM

T_{OFF} = czas stanu niskiego sygnału PWM

T = okres sygnału PWM



Rysunek 6: Wykres czasu T dla napięcia U - sygnał PWM

Na rys. 6 widać wykres obrazujący przykładowy sygnał PWM. Aby obliczyć napięcie średnie sygnału PWN, należy podstawić do poniższego wzoru współczynnik wypełnienia oraz napięcie wyjściowe mikrokontrolera:

$$U_{śr} = k_w * U_1$$

Innymi parametrami PWM to szerokość impulsu (ważny parametr przy sterowaniu serwomechanizmem) oraz częstotliwość impulsów.

Sygnał PWM można wykorzystać do sterowania pozycji ramienia serwomechanizmu,

8 [2] Deshmukh V Ajay (2005), Microcontrollers: Theory and Applications, Tata McGraw-Hill Education, s. 102

ilością obrotów silnika prądu stałego lub jasnością diody elektroluminescencyjnej. W przypadku sterowania jasnością świecenia diody należy odpowiednio dostosować sygnał PWM. W rzeczywistości, dioda świeci się i gaśnie w małych odstępach czasu. By oszukać ludzkie oko, sygnał PWM powinien mieć najmniejszą częstotliwość 50 Hz, by uzyskać efekt płynnego przyciemniania lub rozjaśniania diody. Zmienianie współczynnika wypełniania sygnału PWM będzie wpływać na jasność świecenia diody. Im mniejszy współczynnik, tym mniej światła będzie emitowane przez diodę.

3.4 Szeregowy interfejs SPI

Na rysunku 10 widać rozmieszczenie poszczególnych gniazd. Niektóre z nich mają więcej niż jedną funkcję. Przykładem mogą być piny od 9 do 13, które obsługują interfejs SPI.

SPI (ang. *Serial Peripheral Interface*) jest synchroniczną magistralą wykorzystywaną do komunikacji z urządzeniami peryferyjnymi na niewielkich odległościach. W połączeniu SPI istnieją dwa rodzaje urządzeń: *Master* i *Slave*. Master jest tylko jeden i zazwyczaj jest to mikrokontroler. Nie ma możliwości zmiany ról urządzeń w trakcie ich działania.

SPI korzysta z czterech logicznych linii sygnałowych:

- SCLK (ang. *Serial Clock*) – sygnał z urządzenia master,
- MOSI (ang. *Master Output, Slave Input*) – sygnał z urządzenia mastera do slave,
- MISO (ang. *Master Input, Slave Output*) – sygnał z urządzenia slave do master,
- SS (ang. *Slave Select*) – master ustawia linię SS na stan niski, by wybrać urządzenie, z którym będzie się komunikować.

Linie SCLK, MOSI i MISO są wspólne dla każdego urządzenia. Interfejs SPI umożliwia jednak komunikację z wieloma urządzeniami wykorzystując linię SS. Master wysyła sygnał logiczny niski do modułu peryferyjnego, z którym będzie wymieniał dane. Master może komunikować się tylko z jednym urządzeniem jednocześnie. Urządzenia *slave* nie mogą się bezpośrednio ze sobą komunikować.

Aby rozpocząć komunikację, urządzenie master ustawia generator taktujący na częstotliwość wspieraną przez moduł, z którym będzie wymieniać dane. Następnie wybiera to urządzenie poprzez ustawienie stanu logicznego niskiego na jego linii SS. Moduł korzystający

z magistrali SPI do komunikacji powinien posiadać bufor trójstanowy. Gdy sygnał logiczny na linii SS będzie wysoki, czyli urządzenie nie będzie wybrane do transmisji, linia MISO tego urządzenia będzie rozłączona, by zapobiec zakłóceniom działania innego urządzenia, które może korzystać w danej chwili z tej linii. Transmisja jest dwukierunkowa (pełny dupleks) na każdym cyklu zegarowym SPI. *Master* i *slave* wysyłają jednocześnie dane bit po bicie, nawet jeżeli żądana transmisja jest jednokierunkowa.

Do rozpoczęcia transmisji potrzebne jeszcze są dwa parametry: polaryzacja i faza sygnału taktującego[9]. Polaryzacja linii taktującej (CPOL) jest zależna od stanu tej linii w trybie nieaktywnym. Wartość CPOL jest równa zero, gdy linia w trybie nieaktywnym jest w stanie niskim. Dla wartości równej jeden linia w trybie nieaktywnym jest w stanie wysokim.

Faza sygnału taktującego określana jest jako CPHA. Gdy wartość CPHA jest równa zero, dane są odbierane na zboczu opadającym sygnału taktującego, a wysyłane na zboczu narastającym. Dla CPHA równego jeden, dane są odbierane na zboczu narastającym sygnału taktującego, a wysyłane na zboczu opadającym.

Istnieją cztery tryby SPI, które korzystają z różnych konfiguracji polaryzacji i fazy sygnału:

Tabela 1: Lista trybów SPI

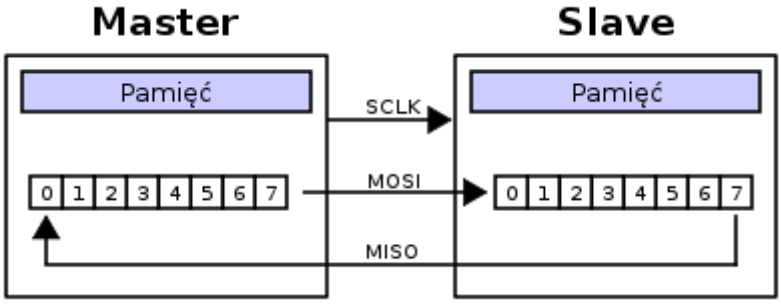
Tryb	**Polaryzacja (CPOL)**	**Faza (CPHA)**
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

Transmisja zazwyczaj wymaga dwóch rejestrów przesuwnych o odpowiedniej dla danych urządzeń długości słowa. Zazwyczaj są to rejestry przesuwne ośmiobitowe SISO (wyjście i wejście szeregowe). Dane są przesuwane z rejestru zaczynając od najbardziej znaczącego bitu, a nowe dane do rejestru są przesuwane od najmniej znaczącego bitu (rys. 7). Po zakończeniu wymiany danych z rejestrów, urządzenia wymieniają się ich wartością. Jeżeli transmisja nie jest zakończona i następna porcja danych musi zostać wysłana, rejestry są

9 Introduction to I²C and SPI protocols, [11] http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/?/article/AA-00255/22/Introduction-to-SPI-and-IC-protocols.html [dostęp 02.04.2016]

przeładowywane. Transmisja może trwać przez dowolną liczbę cyklów zegara, a po zakończeniu linia SS ustawiana jest na stan logiczny wysoki.

SPI jest standardem de facto, ale różne sposoby konfiguracji trybów SPI i długości słowa powodują, że większość urządzeń posiada swój własny protokół SPI.



Rysunek 7: Dwa rejestry przesuwne tworzące bufor cykliczny, wykorzystujące do transmisji magistralę SPI

Aby zaprogramować mikontroler Arduino do komunikacji poprzez SPI, należy zaimportować bibliotekę SPI.h w Arduino IDE. Następnie trzeba wybrać odpowiednie parametry komunikacji funkcją „*SPISettings()*”, pokazane w Listingu 1.

```
1. #include „SPI.h”
2. SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI_MODE0))
```

Listing 1: Import biblioteki SPI oraz inicjalizacja

Pierwszy parametr ustala maksymalną prędkość transmisji SPI w Hz. Arduino będzie automatycznie wybierać najlepszą prędkość dla urządzeń, która jest równa lub mniejsza wybranej. W przypadku problemów z poprawną komunikacją, prędkość należy zmniejszyć.

Drugi parametr ustawia sposób przesuwania danych w rejestrach. Niektóre urządzenia zaczynają przesuwanie od najmniej znaczącego bitu (LSB), inne od najbardziej znaczącego bitu(MSB). Ustawienie to można modyfikować podając parametr MSBFIRST lub LSBFIRST. Trzeci i ostatni parametr określa tryb transmisji SPI (tabela 1).

Następnym krokiem jest ustawienie pinu w tryb wyjścia, który będzie wykorzystywany jako SS dla wybranego urządzenia, którego stan początkowy będzie wysoki.

Oznacza to, że komunikacja z wybranym urządzeniem będzie w tym momencie nieaktywna. Następnie aktywujemy magistralę SPI.

Aby wysłać dane do wybranego urządzenia, należy ustawić jego linię SS na stan niski, a dane wysyłać wywołaniem funkcji, która zwraca otrzymane dane, tak jak w Listingu 2.

```
1. int pinSS = 8;
2. pinMode(pinSS, OUTPUT);
3. digitalWrite(pinSS, HIGH);
4.
5. SPI.begin();
6.
7. digitalWrite(pinSS, LOW);
8. otrzymane_dane = SPI.transfer(bajt);
```

Listing 2: Tranfer danych poprzez SPI

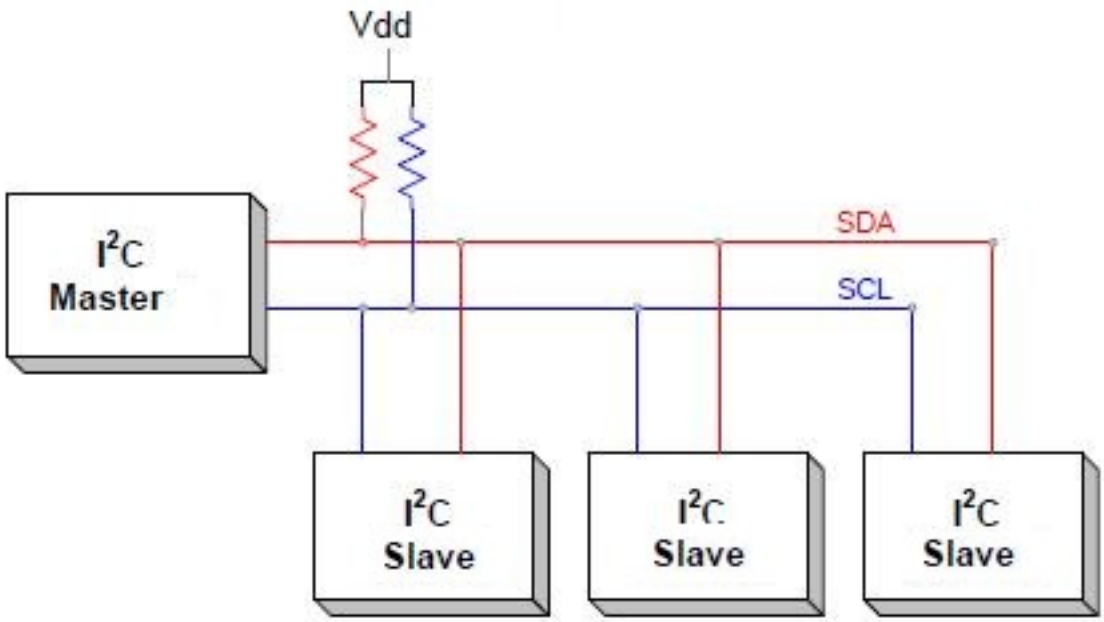
3.5 Magistrala I2C/TWI

I^2C (ang. Inter-Integrated Circuit/Two wire interface, zwana też IIC) jest drugą magistralą obsługiwaną przez Arduino Uno, która służy do komunikacji z urządzeniami peryferyjnymi. Głównie wykorzystywana jest do komunikacji z urządzeniami, które nie wymagają wysokich prędkości transmisji, a nastawione są na prostotę i niski koszt budowy. Przykładem mogą być: odczyt danych z zegara RTC, odczyt czujników różnego rodzaju czujników, sterowanie diodami LED, wyświetlanie na ekranach LCD, pamięć EEPROM.

IIC wykorzystuje tylko dwa przewody do komunikacji: SDA i SDL. Są one podłączone poprzez rezystory podciągające w górę (ang. *pull-up*). Standardowe napięcie zasilające to 5 V lub 3.3 V. Linią SDA przesyłane są dane w dwóch kierunkach, a linią SDL impulsy zegarowe, które synchronizują transmisję danych. Podłączone urządzenia, podobnie jak w SPI, dzielą swoje role na jednego master i wiele slave, jednak istnieje możliwość ich ponownej konfiguracji i wybranie innego mastera w nowej sesji transmisji(po wysłaniu polecenia STOP)[10]. Na rysunku 8 widać przykładową konfigurację z jednym urządzeniem *master* i wieloma urządzeniami *slave.* Dowolne urządzenie podłączone do magistrali IIC może zainicjować transmisję, co uczyni je masterem danej sesji.

10 [3] Eady Fred (2004), Networking and Internetworking with Microcontrollers, Elsevier, rodział 6

Zamiast linii SS, IIC korzysta z 7-bitowej lub 10-bitowej przestrzeni adresowej, by wysyłać dane do odpowiednich urządzeń.

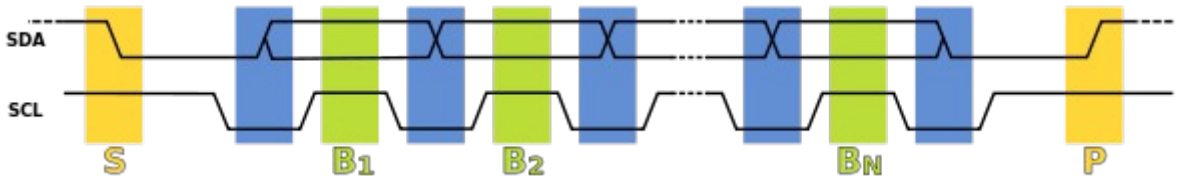


Rysunek 8: Przykładowa konfiguracja magistrali IIC

Aby rozpocząć transmisję, urządzenie z rolą master ustawia na linii SDA stan niski, generując tym samym polecenie startu. Polecenie startu generowane jest poprzez przejście linii SDA ze stanu wysokiego na stan niski, podczas gdy linia SCL jest ustawiona na stan wysoki. Następnie wysyła osiem bitów słowa adresu. Pierwsze siedem bitów jest adresem urządzenia z rolą slave, z którym master chce się wymienić danymi, a ostatni, ósmy bit wybiera rodzaj transmisji: zero jako zapis i jeden jako odczyt danych[11]. Jeżeli urządzenie o wybranym adresie istnieje, to po zakończeniu wysyłania żądania ze słowem adresowym, urządzenie to wymusza na linii SDA stan niski, potwierdzając w ten sposób przyjęcie żądania transmisji (bit ACK). Dalsze etapy transmisji przebiegają analogicznie – pierwsze osiem bitów to transmitowane synchronicznie z impulsami linii SCL dane, a w trakcie dziewiątego impulsu następuje wysłanie potwierdzenia przez urządzenie slave. Wszelkie dane, oprócz polecen start i stop, transmitowane są w czasie, gdy linia SCL ustawiona jest na stan logiczny niski. Polecenie stopu kończy transmisję. Stop generowany jest poprzez przejście linii SDA ze stanu niskiego na stan wysoki, podczas gdy linia SCL jest ustawiona na stan wysoki. Na rysunku 9 widać transmisję kolejnych bajtów B_n, gdzie S oznacza polecenie *START*, a P

11 Ibidem

oznacza polecenie *STOP*.



Rysunek 9: Transmisja I^2C

Aby wykorzystać magistralę I^2C w Arduino, potrzebna jest biblioteka „Wire".

```
1. #import „Wire.h";
2. Wire.begin();
3. Wire.beginTransmission(0x27);
4. Wire.write(dane);
5. Wire.endTransmission();
```

Listing 3: Wysyłanie danych poprzez IIC

Po aktywacji magistrali, dane wysyłane są po jednym bajcie (osiem bitów). Aby przesłać wybrany bajt danych, należy zainicjować połączenie z urządzeniem o wybranym adresie fizycznym (np. 0x27 dla ekranu LCD), wysłać bajt, a następnie zakończyć połączenie, co pokazano w Listingu 3.

Podobnie jest w przypadku odbierania danych, ale trzeba podać ilość żądanych bajtów danych, tak jak w Listingu 4.

```
1. Wire.beginTransmission(0x27);
2. Wire.requestFrom(0x27, 4);
3. otrzymaneDane = Wire.read();
4. Wire.endTransmission();
```

Listing 4: Odbieranie danych poprzez IIC

3.5 Obsługa przerwań

Przerwanie jest sygnałem powodującym zmianę przepływu sterowania, niezależnie od wykonywanego aktualnie programu. Przerwanie wstrzymuje wykonywany program i zostaje uruchomiona procedura obsługi przerwania. Po wykonaniu kodu obsługującego przerwanie, następuje powrót do wcześniej wykonywanego programu. Istnieją dwa rodzaje przerwań sprzętowych. Przerwania zewnętrzne, które pochodzą od urządzeń zewnętrznych takich jak czujniki czy klawiatura. Przerwanie wewnętrzne, nazywane też wyjątkami, występuje w sytuacjach wyjątkowych np. wystąpienie błędu wykonywania programu.

Przerwania wykorzystywane są w Arduino, by zautomatyzować działanie mikrokontrolera i usprawnić zarządzanie zegarami[12]. Przerwania wewnętrzne w Arduino wykorzystywane są m.in. do sterowania komunikacją poprzez port szeregowy, mierzenia czasu czy zliczania impulsów. Przerwania wewnętrzne w Arduino wykorzystywane są m.in. w projektach wymagających dużej dokładności przy mierzeniu czasu. Przerwania zewnętrzne wykorzystywane są do komunikacji z zewnętrznymi modułami. W Arduino Uno R3 można wykorzystać do tego celu dwa piny cyfrowe: drugi i trzeci. Aby wykorzystać wybrany pin jako źródło przerwania, należy go „podpiąć” poprzez funkcję:

attachInterrupt(digitalPinToInterrupt(3), funkcja, RISING);

Pierwszym parametrem jest pin źródłowy. Funkcja „*digitalPinToInterrupt”* przekształca numer pinu na numer przerwania (0 lub 1). Drugi parametr wskazuje na funkcję, która zostanie wykonana podczas wystąpienia przerwania (np. wysłanie sygnału z czujnika ruchu). Ostatni parametr określa warunek powstanie przerwania – przerwanie nastąpi jeżeli wykryta zostanie wybrana zmiana w sygnale. Istnieje pięć możliwości wygenerowania przerwania:

- HIGH - przerwanie zostanie wywołane, gdy na pinie pojawi się logiczny stan wysoki,
- LOW - przerwanie zostanie wywołane, gdy na pinie pojawi się logiczny stan niski,
- CHANGE - przerwanie zostanie wywołane, gdy sygnał na pinie zmieni się ze stanu, który, występował przy inicjalizacji, oraz przy każdej kolejnej zmianie,

12 AttachInterrupt(), [1] https://www.arduino.cc/en/Reference/AttachInterrupt [dostęp 09.02.2016]

- RISING - przerwanie zostanie wywołane, gdy sygnał na pinie zmieni się z niskiego na wysoki,
- FALLING - przerwanie zostanie wywołane, gdy sygnał na pinie zmieni się z wysokiego na niski

Istnieje możliwość odłączenia przerwania w przypadku gdy jest potrzeba zastosowania innego rodzaju przerwania, lub inną funkcję wywoływaną przez to przerwanie. Do tego celu służy funkcja „*detachInterrupt()”*. Istnieją również funkcje pozwalające globalnie sterować systemem przerwań. Jeżeli jakiś fragment programu jest krytyczny dla działania całego urządzenia, można tymczasowo wyłączyć, a poźniej włączyć obsługę przerwań poprzez „*noInterrupts()”* i „i*nterrupts()”*. Zmienne, które wykorzystywane są podczas obsługiwania przerwania, należy podczas ich deklaracji poprzedzić słowem „*volatile”*. Jest to informacja dla miktokontrolera, że dana zmienna musi zostać odświeżona po zakończeniu obsługi przerwania – została ona zmieniona podczas jego wykonywania.

Przykładowym zastosowaniem może być licznik rowerowy. Podczas jazdy koło może wykonywać kilkanaście obrotów na sekundę. Kontaktron podłączony do mikrokontrolera wysyła impuls za każdym razem, gdy magnes zamknie obwód. Gdy mikrokontroler odbierze sygnał z kontaktronu, oblicza on aktualną prędkość i przejechany dystans, a następnie wyświetla wynik na ekranie LCD. Jednak przy dużej liczbie obrotów koła, mikrokontroler może nie przetworzyć części impulsów. Wiąże się to z koniecznością wykonania obliczeń – podczas gdy mikrokontroler oblicza prędkość lub przejechaną odległość, nie jest w stanie jednocześnie odczytać zmiany na wejściu. Aby temu zapobiec, kontaktron należy podłączyć do wejścia obsługującego przerwania. W ten sposób każdy impuls zostanie odczytany, a wykonywane w momencie wysłania sygnału obliczenia zostaną na krótką chwilę wstrzymane.

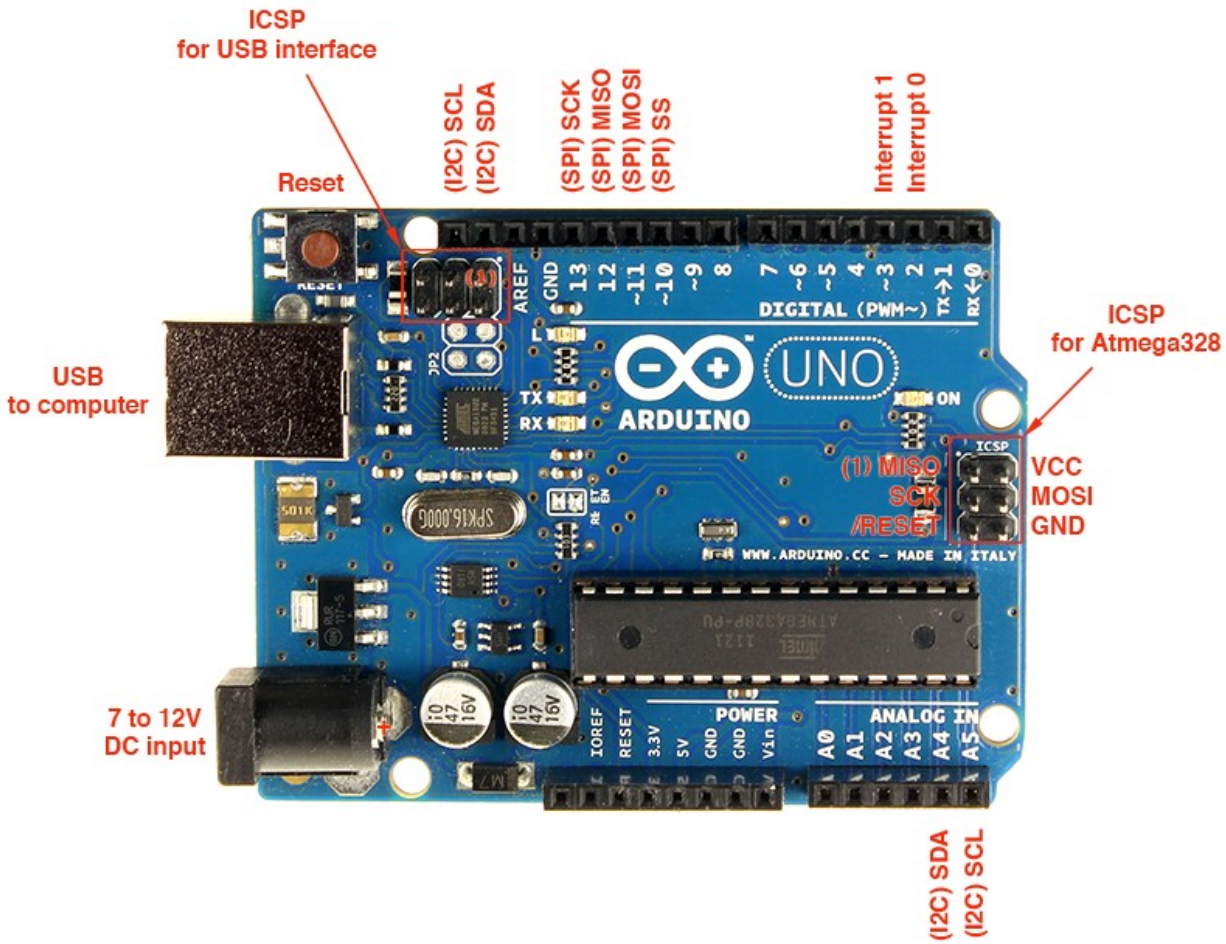
4. Wykorzystane podzespoły

W tym rozdziale przedstawiono wykorzystane w projekcie moduły oraz sposób ich podłączenia. Opis każdego elementu zawiera jego szczegółowe parametry, ogólne zasady działania, schematy oraz przykłady wykorzystania w Arduino, wraz z kodem źródłowym i sposobem podłączenia.

4.1 Arduino Uno R3

Głównym elementem urządzenia jest chiński klon Arduino Uno R3, który jest w pełni kompatybilny z oprogramowaniem Arduino IDE oraz posiada pełną funkcjonalność oryginału. Wykorzystuje on mikrokontroler ATmega328P. Posiada czternaście cyfrowych pinów wejścia/wyjścia, sześć pinów analogowych, kryształ kwarcowy 16 MHz, złącze USB, złącze zasilające, złącze ICSP i przycisk resetujący mikrokontroler[13]. Wymagane napięcie zasilające mieści się w zakresie 7 – 12V, jednak można zasilać go poprzez port USB pod warunkiem, że nie podłączymy dużej liczby urządzeń peryferyjnych. Dodatkowo, Arduino Uno posiada resetowany bezpiecznik polimerowy, który uchroni złącze USB przed zwarciem lub przeciążeniem, gdy podłączymy za dużo modułów. Na rysunku 10 zostały oznaczone wszystkie wejścia i złącza, oraz wykorzystywane na nich magistrale.

13 Arduino UNO & Genuino UNO,[2] https://www.arduino.cc/en/Main/ArduinoBoardUno [dostęp 03.02.2016]



Rysunek 10: Diagram złącz Arduino Uno

Arduino Uno programuje się wykorzystując Arduino IDE. Wystarczy podłączyć je przez USB, wybrać model z listy i wgrać napisany program. Mikrokontroler ATmega328P jest fabrycznie zaprogramowany „bootloaderem”, czyli programem rozruchowym, co pozwala programować Arduino bez wykorzystania zewnętrzenego programatora. Jednak wykorzystanie takiego programatora też jest możliwe. Do tego celu można wykorzystać złączę ICSP (ang. In-Circuit Serial Programming). Rozwiązanie to umożliwia zaprogramowanie mikrokontrolera bez konieczności demontażu, np. aktualizację oprogramowania.

Pełna specyfikacja techniczna Arduino Uno[14]:

Tabela 2: Specyfikacja Arduino Uno R3	
Mikrokontroler	ATmega328P
Napięcie pracy	5V
Napięcie wejściowe(zalecane)	7-12V
Napięcie wejściowe(limit)	6-20V
Cyfrowe piny WE/WY	14 (z których 6 to PWM)
Cyfrowe piny PWM	6
Piny Analogowe	6
Natężenie DC na pinie cyfrowym	20 mA
Natężenie DC dla pinu 3.3V	50 mA
Pamięć Flash	32 KB (ATmega328P) 0.5 KB wykorzystane przez bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Taktowanie zegara	16 MHz
Długość	68.6 mm
Szerokość	53.4 mm
Waga	25 g

4.2 Czytnik kart RFID

W pracy wykorzystany został czytnik kart RFID, który odczyta ID zbliżonej do niego karty, i wyśle go poprzez magistralę SPI do mikrokontrolera. Następnie, również wykorzystując magistralę SPI, sprawdzi, czy odczytany numer karty istnieje na karcie micro SD jako karta białej listy. Jeżeli dany numer ID rzeczywiście znajduje się na białej liście, dostęp zostanie przyznany, a drzwi zostaną odblokowane.

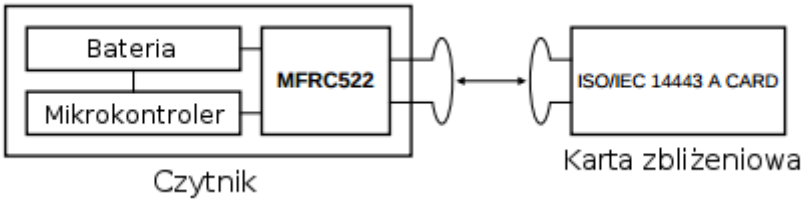
14 Ibidem

Wykorzystany do tego celu czytnik to MFRC522 (rys. 11) – układ scalony do bezprzewodowej komunikacji na częstotliwości 13.56 MHz.

Rysunek 11: MFRC522

Wbudowany w układ transmiter jest w stanie komunikować się z kartami zgodnymi ze standardem ISO/IEC 14443 A/MIFARE, zapewniając implementację dla demodulacji i dekodowania sygnału z tego typu kart (rys. 12). Układ umożliwia kontrolę parzystości i błędów CRC (ang. *Cyclic Redundancy Check* – cykliczna kontrola nadmiarowa). MFRC522 umożliwia komunikację z kartami MIFARE z maksymalną szybkością 848 kilobodów, jako pełny dupleks[15].

15 Dokumentacja techniczna MFRC522, [7] http://www.nxp.com/documents/data_sheet/MFRC522.pdf [dostęp 20.01.2016]



Rysunek 12: Tryb odczytu/zapisu

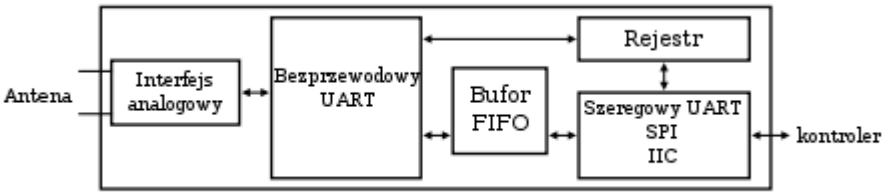
Układ wspiera trzy rodzaje komunikacji z mikrontrolerem:

- szeregowy interfejs SPI,
- interfejs UART,
- interfejs I^2C

Jednak w momencie pisania w pracy, w bibliotece dla Arduino została zaimplementowana jedynie możliwość komunikacji z układem MFRC522 poprzez interfejs SPI[16].

Najważniejsze parametry czytnika:

- analogowy interfejs do demodulacji i dekodowania odpowiedzi,
- możliwość podłączenia dodatkowej anteny,
- standardowa odległość dla odczytu/zapisu na kartach to 50 mm,
- maksymalna szybkość komunikacji z kartami - 848 kilobodów,
- maksymalna prędkość transmisji dla SPI – 10 Mbit/s,
- napięcie zasilające – od 2.5 V do 3.3 V,
- koprocesor CRC,
- programowalne piny wejścia/wyjścia



Rysunek 13: Uproszczona budowa MFRC522

Zadaniem interfejsu analogowego jest modulacja i demodulacja sygnałów analogowych

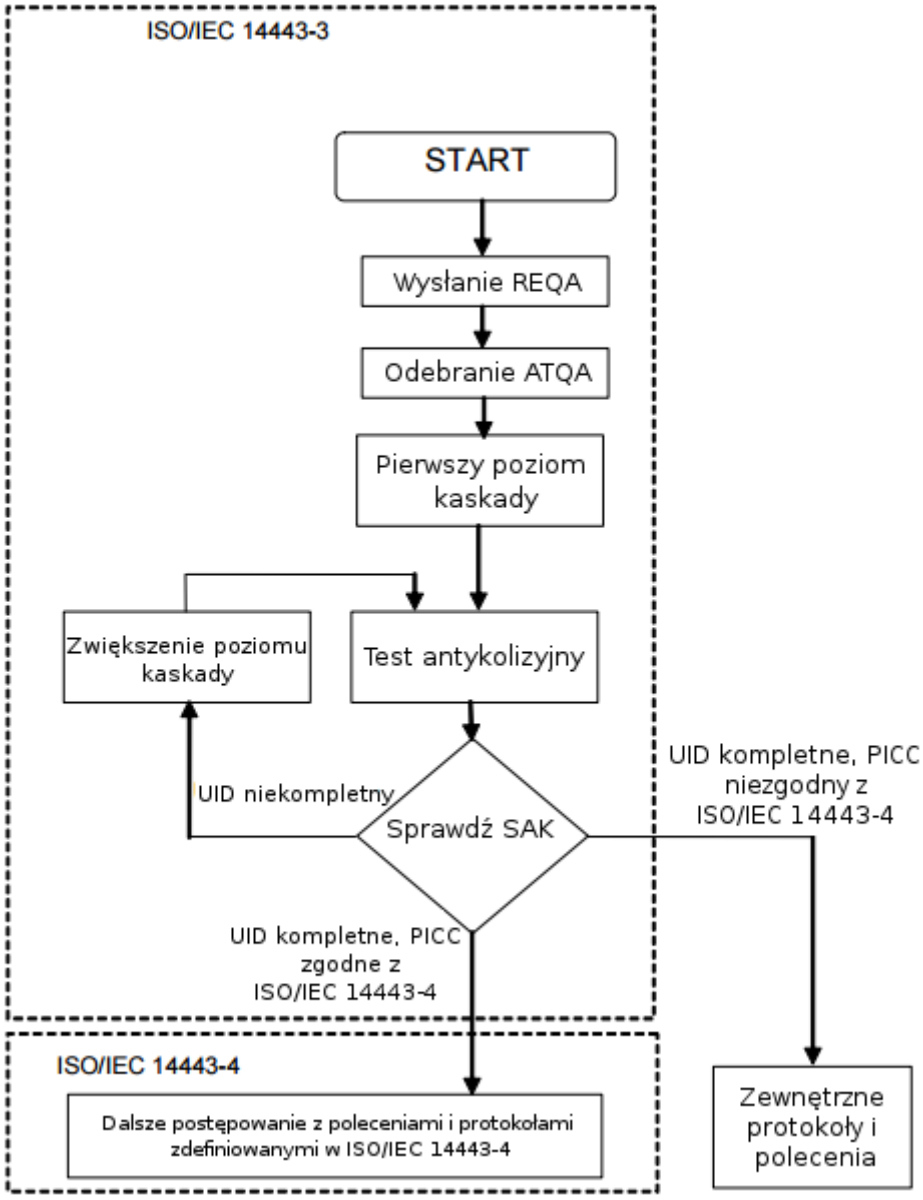
16 MFRC52, [12] http://playground.arduino.cc/Learning/MFRC522 [dostęp 02.04.2016]

pochodzących z anteny. Bezprzewodowy interfejs UART zarządza wymaganymi protokołami transmisji danych do kontrolera. Bufor FIFO zapewnia szybką i prostą transmisję danych pomiędzy mikrokontrolerem a bezprzewodowym interfejsem UART i odwrotnie. Układ MFRC522 zapewnia trzy protokoły do komunikacji z urządzeniem zewnętrznym (rys. 13).

Czytnik umożliwia zarówno odczyt jak i zapis danych na karcie zgodnej z obsługiwanym standardem. W praktycznej części pracy wykorzystana została jednak tylko możliwość odczytu unikalnego numeru identyfikacyjnego UID (ang. *Unique Identifier*) karty PICC (ang. *Proximity Card* – karta zbliżeniowa).

Proces uzyskania UID rozpoczyna się od wykrycia karty znajdującej się w polu operacyjnym czytnika PCD (ang. *Proximity Coupling Device* – czytnik kart zbliżeniowych*)*. Czytnik wysyła powtarzające komendy *Request* dla dwóch typów kart: Typu A oraz Typu B (rys. 14). Głównymi różnicami pomiędzy tymi kartami są metody modulacji, sposoby kodowania danych i procedurami wywołania protokołu. Częścią wspólną jest protokół transmisji danych. PCD wysyła komendy *REQA* oraz *REQB* w dowolnej sekwencji dla stałego lub konfigurowalnego współczynnika wypełnienia[17].Aby karta, która wymaga 5ms na wysłanie odpowiedzi, mogła zostać wykryta, czytnik musi emitować pole elektromagnetyczne przez przynajmniej 5.1 ms.

17 ISO/IEC FCD 14443-3, s. 5



Rysunek 14: Procedura inicjalizacji karty

Wszelkie karty PICC w trybie czuwania *IDLE* wysyłają odpowiedź ATQA (ang *Answer To Request)*. Czytnik powinien wykryć kolizję w przypadku istnienia więcej niż jednej tego typu karty. Karta po odebraniu komendy wysyła dwa bajty jako odpowiedź, która zawiera ramkę z rozmiarem numeru UID oraz ramkę antykolizyjną[18]. Po odebraniu odpowiedzi, czytnik wykonuję pętlę antykolizyjną dla kaskady pierwszego poziomu i wysyła komendę *SELECT*

18 Ibidem, s. 17

do wykrytej karty, a karta przechodzi w tryb gotowości *READY*. Poziom kaskady zależy od rozmiaru UID. Dla 4 bajtów – poziom 1.
Dla 7 bajtów – poziom 2. Dla 10 bajtów – poziom 3[19]. Dla każdego poziomu sprawdzana jest odpowiedź karty SAK(ang. *Select AcKnowledge*), która zawiera informację, czy został wysłany kompletny numer UID. Jeżeli nie, sekwencja powtarzana jest dla zwiększonego poziomu kaskady, aż do momenty uzyskania kompletnego numeru UID. Następnie karta przechodzi z trybu *READY* do trybu aktywnego *ACTIVE,* w którym może przyjąć kolejne polecenia od czytnika PCD, takie jak polecenie odczytu lub zapisania danych na karcie.

Do pracy z MFRC522 można wykorzystać bibliotekę RFID dostępną pod adresem https://github.com/miguelbalboa/rfid. Biblioteka oferuje implementację protokołu SPI oraz obsługę kart zbliżeniowych typu A. Karty typu B nie są obsługiwane przez czytnik, dlatego też ich obsługa nie jest uwzględniona w bibliotece.

Aby zacząć pracę z czytnikiem, należy zacząć od importowania biblioteki. Następnie tworzymy instancję klasy czytnika, definiując piny RST oraz SS czytnika RFID, oraz inicjujemy czytnik. Całość widoczna jest w Listingu 5.

```
1. #include <MFRC522.h>
2. #define RST_PIN  6
3. #define SS_PIN  7
4. MFRC522 Rfid(SS_PIN, RST_PIN);
5. Rfid.PCD_Init();
```

Listing 5: Inicjalizacja czytnika kart RFID

Aby odczytać numer ID karty, wysyłamy w pętli żądanie do kart znajdujących się w zasięgu czytnika. Jeżeli funkcja nie otrzyma odpowiedzi, pętla zostaje przerwana i wykonuje się od początku. Po wykryciu karty następuje odczytanie numeru identyfikacyjnego do zmiennej, które zostało przedstawione w Listingu 6.

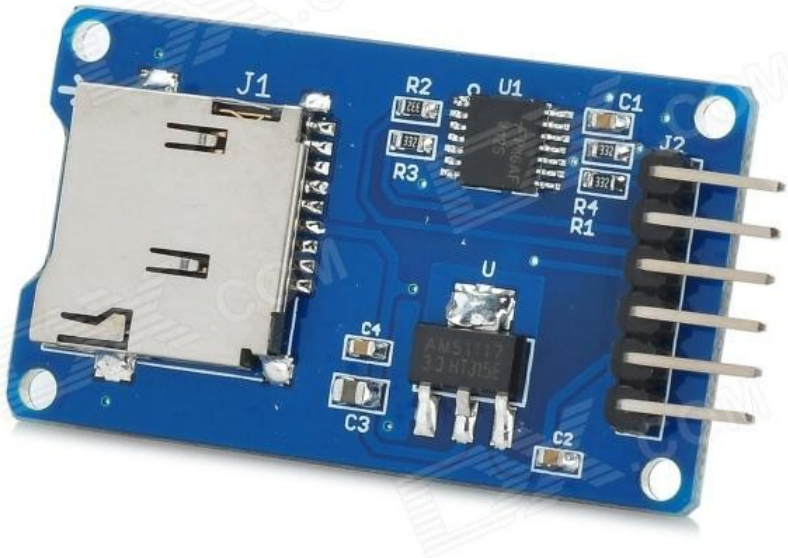
19 Ibidem, s. 21

```
1. if ( ! Rfid.PICC_IsNewCardPresent()) {
2.         return false;
3. }
4. if ( ! Rfid.PICC_ReadCardSerial()) {
5.         return true;
6.  }
7.  byte cardID[4];
8.  for (int i = 0; i < 4; i++) {
9.         cardID[i] = Rfid.uid.uidByte[i];
10.          }
```

Listing 6: Odczytywanie numeru ID karty RFID

4.3 Adapter kart MicroSD Catalex

Dane dotyczące odczytanych kart i aktywności urządzenia zapisywane są jako dzienniki w plikach tekstowych na karcie MicroSD. Do tego celu wykorzystałem moduł adaptera kart MicroSD firmy Catalex (rys. 15).



Rysunek 15: Adapter MicroSD

Moduł ten posiada następujące parametry:

- obsługa kart MicroSD i MicroSDHC
- napięcie zasilające 5 V
- regulator napięcia do 3.3 V
- komunikacja poprzez interfejs SPI

Do modułu nie jest dołączana dokumentacja techniczna, nie znajduje się ona też w internecie. Moduł wykorzystany w pracy jest wczesną wersją – v1.0 z roku 2013, z czym wiąże się problem z kompatybilnością z innymi modułami. Po wielu próbach podłączenia jednocześnie wyżej przedstawionego adaptera oraz czytnika kart zbliżeniowych okazało się, że adapter nie wyłącza swojej linii MISO nawet gdy sygnał wysyłany na pin CS(ang. *Chip select*, to samo co *Slave Select*) jest niski. Stale aktywna linia MISO zakłóca pracę pozostałych urządzeń korzystających z magistrali SPI. Aby rozwiązać ten problem, wykorzystałem do tego celu bufor trójstanowy SN74HC125N. Wystąpił również problem z wbudowaną w Arduino IDE biblioteką do obsługi kart SD. Została ona zbudowana na podstawie starej, nieaktualnej wersji „sdfatlib" Williama Greimana, która nie jest w pełni kompatybilna z nowym sprzętem i oprogramowaniem. Program napisany z wykorzystaniem tej biblioteki generował błędy i niemożliwe było poprawne zaprogramowanie modułu. Rozwiązaniem problemu było wykorzystanie najnowszej wersji biblioteki tego samego autora, która zachowując pełną kompatybilność ze starą wersją dostępną w Arduino IDE, rozwiązuje wiele problemów z kompatybilnością i wprowadza wiele nowych funkcjonalności[20]. Biblioteka ta posiada rozbudowaną dokumentację techniczną, co ułatwia w dużym stopni zapoznanie się z jej możliwościami.

Aby zapisywać i odczytywać dane z karty MicroSD, należy zacząć od pobrania, zainstalowania i importowania biblioteki w projekcie, deklaracji odpowiednich zmiennych i definicji pinu CS. Następna jest inicjalizacja adaptera dla wybranego pinu CS wraz z parametrem SPI_HALF_SPEED, który poprawia stabilność pracy adaptera, co ukazano w Listingu 7.

20 Arduino FAT16/FAT32 Library, [2] https://github.com/greiman/SdFat [dostęp 06.02.2016]

```
1. #include <SdFat.h>
2. SdFat sd;
3. SdFile plik;
4. const int sd_cs = 8;
5. sd.begin(sd_cs, SPI_HALF_SPEED)
```

Listing 7: Inicjalizacja czytnika kart microSD

Aby wyświetlić zawartość pliku tekstowego, należy postępować tak jak w Listingu 8.

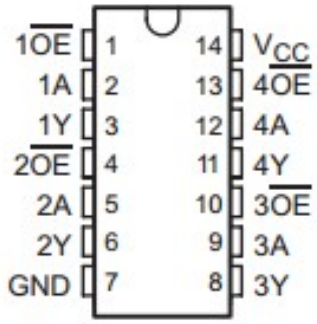
```
1. if (plik.open(filename, O_READ)) {
2.         while (plik.available())
3.         {
4.                 char ch = plik.read();
5.                 Serial.print(ch);
6.         }
7. plik.close()
```

Listing 8: Odczytanie danych z pliku

Najpierw otwieramy plik o wybranej nazwie w trybie do odczytu. Jeżeli plik został otwarty poprawnie, zaczynamy odczytywać zawartość. Funkcja *available* zwraca ilość bajtów jaka pozostała od aktualnej pozycji kursora do końca pliku. Jeżeli ta wartość jest większa niż zero, zapisujemy odczytany bajt do zmiennej wykorzystując funkcję *read*, a następnie wyświetlamy odczytaną wartość. Czynność jest powtarzana do momentu osiągnięcia końca pliku. Na końcu zamykamy otwarty przez nas plik. Jednocześnie można pracować tylko na jednym otwartym pliku.

4.4 Bufor trójstanowy SN74HC125

Bufor trójstanowy, widoczny na rysunku 16, został wykorzystany do odłączania linii MISO czytnika kart MicroSD, gdyż moduł ten nie wyłącza swojej linii MISO w żadnej sytuacji, niezależnie od stanu linii CS. Bufor trójstanowy działa jak przełącznik sterowany elektrycznie – w zależności od sygnału na wejściu OE, bufor przekazuje lub nie przekazuje stanu wejścia A na wyjście Y.



Rysunek 16: SN74HC125

Gdy na wejściu OE będzie stan wysoki, wyjście Y będzie w stanie wysokiej impedancji, efektywnie odłączając sygnał z wejścia A. Napięcie robocze dla bufora mieści się w zakresie od 2 V do 6 V.

4.5 Serwomechanizm TowerPro SG90

W pracy został wykorzystany serwomechanizm TowerPro SG90, widoczny na rysunku 17. Po zbliżeniu karty do czytnika RFID, po sprawdzeniu czy ID karty zbliżeniowej znajduje się na białej liście na karcie MicroSD, następuje otwarcie zamka poprzez obrót wału serwomechanizmu.

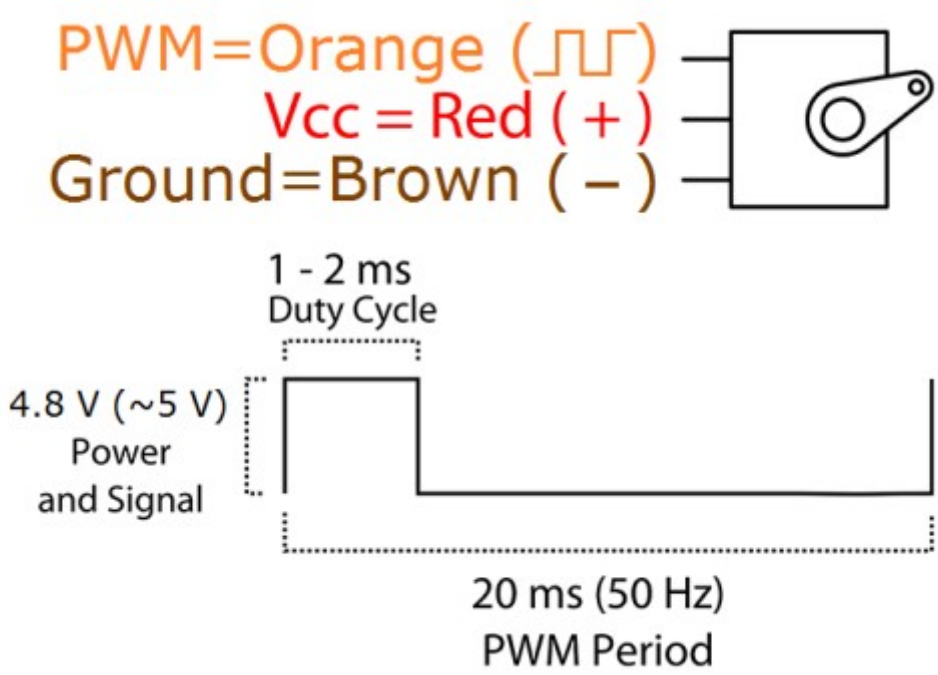
Parametry serwomechanizmu są następujące:
- moment: 1,8 kg/cm,
- waga: 9g,
- prędkość obrotu: 0,12 s/60st,
- napięcie pracy: 4,5V - 6V

Rysunek 17: Serwomechanizm TowerPro SG90

Serwomechanizm zbudowany jest z silnika prądu stałego, przekładniami i układu sterującego. Na zewnątrz obudowy wychodzi wał, którego pozycja kątowa ustalana jest przez odpowiedni sygnał. Utrzymanie tego samego sygnału spowoduje utrzymanie wału w stałej pozycji. Pozycja wału jest zawsze znana poprzez zastosowanie potencjometru obrotowego, który obraca się razem z wałem. Serwomechanizm sterowany jest sygnałem analogowym. Aby sterować serwomechanizmem przy wykorzystaniu mikrokontrolera, należy wykorzystać modulację szerokości impulsów. Do tego celu w Arduino możemy wykorzystać piny 11, 10, 9, 6, 5 i 3.

Serwomechanizm podłącza się poprzez trzy przewody: przewód sygnałowy i dwa zasilające (5V). Pozycja serwa nie jest zależna od współczynnika wypełnienia, a ustawiana jest na podstawie długości impulsu. Serwo oczekuje na impuls co 20 ms. Pozycja neutralna

dla serwa, to taka, dla której pozostały obrót zarówno w kierunku zgodnym, jak i przeciwnym do ruchu wskazówek zegara, jest taki sam. Aby ustawić serwo na pozycji neutralnej i utrzymać je na tej pozycji, należy wysłać impuls o szerokości 1.5 ms[21] i wysyłać go ciągle co 20 ms (rys. 18).



Rysunek 18: Parametry sygnału sterującego PWM dla serwomechanizmu

Aby w łatwy sposób kontrolować serwomechanizm podłączony do Arduino, można wykorzystać bibliotekę wbudowaną w Arduino IDE.

```
1. #include <Servo.h>
2. Servo servo;
3. servo.attach(9);
4. servo.write(60)
5. servo.detach();
```

Listing 9: Obsługa serwomechanizmu

Tworzymy obiekt dla serwomechanizmu i go „podłączamy” poprzez funkcję *attach.*

21 Specyfikacja techniczna TowerPro SG90, [14] http://datasheet.sparkgo.com.br/SG90Servo.pdf [dostęp 10.01.2016]

Funkcją „*write()”* możemy ustalić pozycję wału od 0 do 180. Przykładowy program przedstawiono w Listingu 9.

4.6 Ekran 1602 LCD z adapterem I²C

Każde urządzenie elektroniczne powinno posiadać możliwość przekazywania informacji swojemu użytkownikowi. Najprostszą metodą na realizację tej funkcjonalności jest wyemitowanie sygnału świetlnego z wykorzystaniem diody LED. W tej pracy wykorzystano do tego celu ekran LCD 1602. Nazwa modułu oznacza ilość znaków, jakie może wyświetlić – 16 znaków na dwóch wierszach. Adapter I2C podłączony jest do ekranu poprzez następujące piny ekranu[22]:

1. Uziemienie
2. VCC (od 3.3V do 5V)
3. Regulacja kontrastu (VO)
4. Wybór rejestru (ang. *Register Select,* RS). RS=0: Polecenie, RS=1: Dane
5. Odczyt/Zapis (ang. *Read/Write*). R/W=0: Zapis, R/W=1: Odczyt
6. Zegar (*Clock Enable*). Aktywacja zobaczem opadającym
7. Bit 0 (Nie używany w operacjach 4-bitowych)
8. Bit 1 (Nie używany w operacjach 4-bitowych)
9. Bit 2 (Nie używany w operacjach 4-bitowych)
10. Bit 3 (Nie używany w operacjach 4-bitowych)
11. Bit 4
12. Bit 5
13. Bit 6
14. Bit 7
15. Anoda diody podświetlenia (+)
16. Kota diody podświetlenia (-)

Adapter I²C korzysta z ekspandera PCF8574, który zwiększa liczbę wejść

22 Dokumentacja techniczna LCD1602, [5] http://www.elecrow.com/download/LCD1602.pdf [dostęp 04.02.2016]

równoległych o 8, wykorzystując tylko dwa piny dla interfejsu szeregowego I^2C[23]. Adapter I^2C umożliwia również sterowanie kontrastem ekranu potencjometrem, oraz wyłączenie podświetlenia ekranu, wyciągając zworkę, umożliwiając jednocześnie sprzętowe kontrolowanie diody podświetlającej.

Aby wykorzystać ekran w Arduino, można skorzystać z biblioteki LiquidCrystal_I2C. Wyświetlanie tekstu na ekranie odbywa się na podobnej zasadzie jak na szeregowym monitorze. Przykładowy kod znajduje się w Listingu 10.

```
1. #include <Wire.h>
2. #include <LiquidCrystal_I2C.h>
3. LiquidCrystal_I2C lcd (0x27, 16, 2);
4. lcd.init (); //inicjalizacja ekranu
5. lcd.backlight (); //włączenie podświetlenia LCD
6. lcd.setCursor(0, 0); //ustawienie kursora na początek ekranu
7. lcd.print(F("Starting...")); //F() zapisuje tekst w pamięci flash zamiast SRAM
```

Listing 10: Przykładowy kod obsługujący ekran LCD

Program należy zacząć od importowania wymaganych bibliotek. Następne jest utworzenie nowej instancji ekranu, o ustalonym adresie na magistrali I^2C i rozmiarze ekranu. Kolejnymi krokami jest inicjalizacja wyświetlacza i włączenie podświetlenie. Tekst można wyświetlać w dowolnym miejscu na ekranie, wykorzystując funkcję *setCursor()*. Istnieje również możliwość wykorzystania własnych znaków do wyświetlania. Rozdzielczość jednego segmentu to 5*7 pikseli o rozmiarze 0.55*0.5 mm.

23 Dokumentacja techniczna PCF8574, [6]
http://www.nxp.com/documents/data_sheet/PCF8574.pdf [dostęp 04.02.2016]

5. Arduino IDE

Arduino IDE jest darmowym środowiskiem programistycznym na licencji open-source. Pozwala w łatwy sposób programować oficjalne płytki z logo Arduino (rys. 19), klony oraz urządzenia kompatybilne z Arduino.

Rysunek 19: Logo Arduino

Oprogramowanie dostępne jest na następujące systemy operacyjne:

- Windows,

- Linux 32 i 64 bit,

- Mac OS X 10.7 Lion lub nowszy

Podczas tworzenia kodu dla urządzenia, którego dotyczy ta praca, wykorzystano wersję oprogramowania Arduino IDE na system Linux Mint 17.3 64 bit. Wiąże się to z przewagą jaką daje Linux dla twórców oprogramowania – najważniejsze narzędzia i sterowniki są wbudowane w system i nie wymagają dodatkowej konfiguracji. W przypadku podłączenia układu Arduino do komputera z systemem Linux nie jest wymagana instalacja dodatkowych sterowników, podczas gdy na systemie Windows jest to wymagane[24].

24 Getting Started with Arduino on Windows, https://www.arduino.cc/en/Guide/Windows [dostęp 10.01.2016]

Instalacja oprogramowania Arduino IDE jest bardzo prosta na systemie Linux Mint 17.3, ponieważ znajduje się one w oficjalnych repozytoriach tego systemu. Instalacja odbywa się poprzez wpisanie jednego polecenia w Terminalu (skrót klawiszowy: Alt+Ctrl+T):

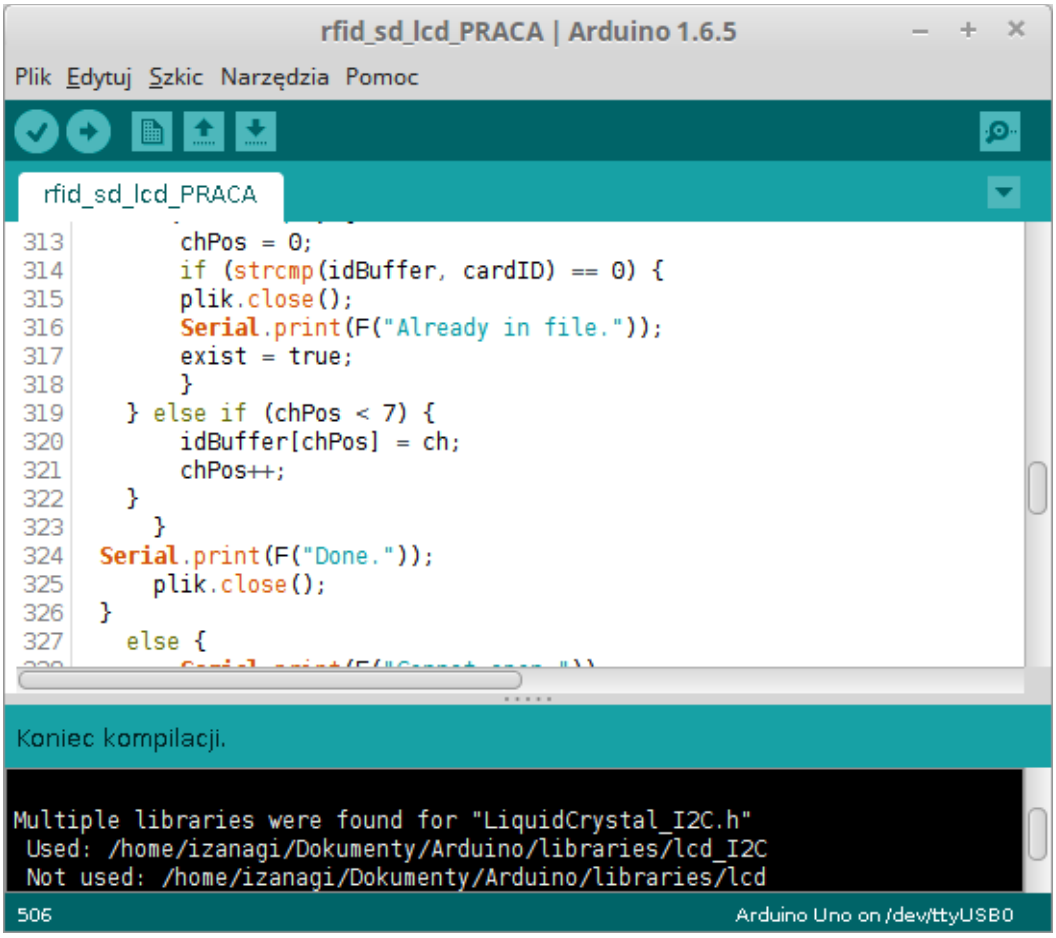
sudo apt-get install arduino

Konieczne będzie wpisanie hasła administratora. Aby umożliwić Arduino IDE korzystanie z komunikacji poprzez port szeregowy, należy dodać aktualnie zalogowanego użytkownika do grupy *dialout*:

sudo usermod -aG dialout $(whoami)

Aby polecenie *usermod* wprowadziło zmiany w systemie, może być wymagane ponowne zalogowanie się do systemu.

Po uruchomieniu powinien pokazać się interfejs widoczny na rys. 20. Programy napisane w Arduino IDE zwaną są szkicami. W pierwszej zakładce menu można utworzyć nowy szkic, zapisać lub otworzyć szkic z pliku. Szkice zapisywane są w formacie „.ino”. Istnieje również możliwość załadowania przykładów dla wbudowanych bibliotek. Przykłady zawierają programy prezentujące podstawowe możliwości płytki Arduino.



Rysunek 20: Interfejs Arduino IDE

Najczęściej używanym przykładem jest program „*blink*”, który głównie służy do testowania, czy płytka została podłączona do komputera poprawnie i czy układ jest sprawny. Ten przykładowy program powoduje włączanie i wyłączanie wbudowanej diody LED co jedną sekundę. Przedstawiony został w Listingu 11.

```
1.  void setup() {
2.          // inicjalizacja pinu 13 jako wyjścia
3.          pinMode(13, OUTPUT);
4.  }
5.  void loop() {
6.           //pin 13 jest podłączony do wbudowanej diody LED
7.           digitalWrite(13, HIGH);   //ustawienie go na poziom wysoki włączy diodę
8.           delay(1000); //zatrzymanie wykonywania kodu na 1000 ms
9.          digitalWrite(13, LOW); //ustawienie pinu na poziom niski, wyłączenie diody
10.                 delay(1000);
11.         }
```

Listing 11: Przykład "blink"

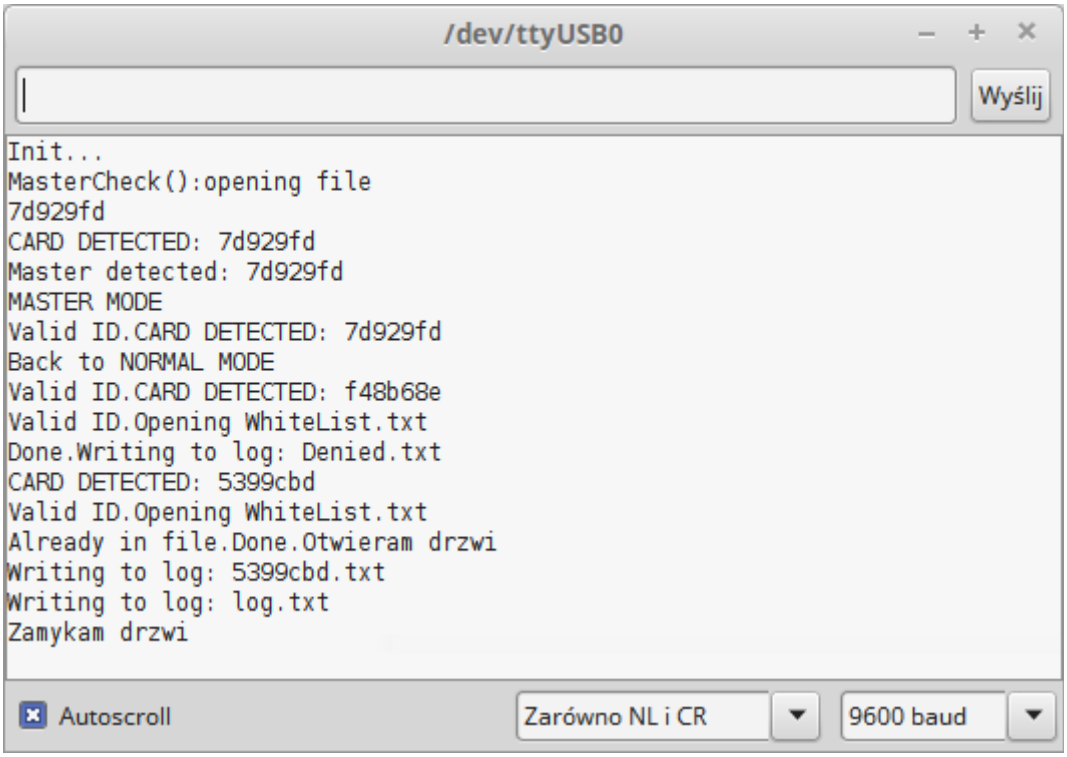
Inne przykłady prezentują możliwości języka: zastosowanie wyrażeń warunkowych, pętli, tablic, operacje na ciągach znaków, komunikacja z komputerem poprzez interfejs szeregowy. Przykłady zawierają również podstawowe techniki programowania układów elektronicznych, np. obsługa przycisków z uwzględnieniem możliwości wystąpienia zakłóceń (tzw. „*debounce”),* odczyt danych z akcelerometru, odczytywanie napięcia na wejściu analogowym czy sterowanie kursorem na ekranie komputera wykorzystując joystick podłączony do Arduino. Program dla Arduino zbudowany jest z dwóch podstawowych funkcji *setup* oraz *loop.* Komentarze w kodzie poprzedza się znakami „//” dla jednego wiersza, lub „/*komentarz*/ ”dla wielu linii. Struktura i przykładowe wykorzystanie komentarzu przedstawiono w Listingu 12.

```
void setup(){
        //funkcja setup uruchamia się tylko raz po uruchomieniu
        //mikrokontrolera
        //w tej funkcji należy umieszczać kod odpowiedzialny za deklaracje
        //zmiennych lub inicjalizację urządzeń
}
void loop{
        /* to jest główna funkcja programu. Kod w niej zawarty jest ciągle
        realizowany w pętli. Realizacja kodu programu może zostać wstrzymana
        poprzez „przerwanie”, które można zrealizować np. poprzez wciśnięcie
        fizycznego przycisku */
}
```

Listing 12: Budowa programu w Arduino IDE

Programowanie w Arduino opiera się głównie na zarządzaniu pinami i wymianą danych z innymi modułami, takimi jak czujniki czy różnego rodzaju wyświetlacze. Dowolny pin cyfrowy może być wyjściem lub wejściem, w zależności od napisanego programu. Jeżeli wybrane wejście zostanie skonfigurowane jako wyjściowe, będzie można ustalić jego stan jako wysoki lub niski. Jeżeli pin będzie ustawiony jako wejściowy, możemy odczytać jego stan i zapisać do zmiennej.

Arduino IDE umożliwia również komunikację z płytką poprzez monitor szeregowy. Pozwala to na wymianę danych z komputerem poprzez złącze USB. Przez rozpoczęciem komunikacji należy wybrać rodzaj wykorzystywanych znaków końca linii oraz prędkość transmisji. Jako znak końca linii można wybrać: nowa linia (ang. *NL – New Line*), powrót karetki (ang. *CR – Carriage Return*) oraz oba równocześnie. Prędkość transmisji można ustawić od 300 do 250000 bodów. Ustawienia te zależą od rodzaju płytki, z którą nastąpi wymiana danych. Dla Arduino Uno R3 znakiem końca linii są NL i CL jednocześnie, a domyślna prędkość transmisji to 9600 bodów. Prędkość można zwiększyć kosztem stabilności. Na rys. 22 monitor wykorzystano do analizy poprawności działania projektu.



Rysunek 21: Monitor szeregowy Arduino IDE

Aby mikrokontroler wyświetlił komunikat na monitorze, należy zainicjować połączenie, określając wybraną prędkość transmisji, tak jak na Listingu 13.

```
1. Serial.begin(9600);
2. Serial.print(F("Start")); //wyświetla tekst bez dodawania znaku końca linii
3. Serial.println(F(" programu")); //wyświetla tekst i dodaje znak końca linii
```

Listing 13: Wykorzystanie połączenia szeregowego

Tekst, który będzie wysłany przez mikrokontroler jest domyślnie przechowywany w pamięci RAM, która jest bardzo ograniczona (2kB dla Arduino Uno). Wyświetlanie dużej ilości wiadomości może szybko zapełnić pamięć mikrokontrolera, powodując jego niestabilną pracę. Aby tego uniknąć, można wykorzystać funkcję F(), która wykorzysta pamięć Flash mikrokontrolera do przechowywania tekstu (32 kB dla Arduino Uno). W ten sposób można zmniejszyć wykorzystanie pamięci RAM kosztem pamięci Flash. Drugą możliwością jest przechowywanie tekstu w pamięci EEPROM Arduino Uno R3, której jest 1kB. Jednak przy projektowaniu programu korzystającego z EEPROM należy pamiętać o ograniczonej ilości operacji zapisu takiej pamięci, która wynosi około 100 000 cykli. Ilość cykli odczytu nie jest ograniczona. Istnieje jeszcze trzecia możliwość przechowywania tekstu, ale jest trudniejsza w realizacji i wymaga dodatkowego modułu – adaptera kart MicroSD. Rozwiązanie to polega na wykorzystania bardzo dużej pamięci, jaką oferują karty microSD. W takim przypadku należy napisać dodatkowe procedury odpowiadające za odczyt danych z karty. Kosztem pamięci Flash mikrokontrolera i jednego dodatkowo wykorzystanego cyfrowego pinu można przechowywać bardzo duże ilości tekstu. Rozwiązanie to jest idealne do projektów, które wykorzystują Arduino jako mały serwer WWW. Pliki stron internetowych czy skrypty można przechowywać na karcie pamięci.

6. Projekt urządzenia

Rozdział ten zawierać będzie ogólny opis i zasady działania ukończonego urządzenia, razem z rysunkami przedstawiającymi urządzenie w trakcie działania. Opisana zostanie budowa pierwszego prototypu oraz omówiony sposób podłączenia modułów do mikrokontrolera. W rozdziale znajdą się również wyjaśnione najważniejsze fragmenty kodu źródłowego, wykorzystanego w projekcie.

6.1 Opis urządzenia, zasady działania

Zwykły zamek w drzwiach może być niewystarczającym zabezpieczeniem w pomieszczeniach, do których dostęp ma wiele osób. Nie ma możliwości sprawdzenia kto i kiedy uzyskał dostęp do zabezpieczonego w ten sposób miejsca. Umożliwienie dostępu dodatkowym osobom wiąże się z koniecznością dorobienia nowych kluczy, co wymaga poświęcenia dodatkowego czasu, i generowaniem dodatkowych kosztów. Zgubienie klucza przez jedną z osób zmniejsza potencjał tego typu zabezpieczenia, umożliwiając dostęp osobom, które tego dostępu nie powinni uzyskać. W przypadku nieautoryzowanego dostępu, sprawdzenie czasu dokonania go jest niemożliwe. Każda osoba posiadająca klucz może również zapomnieć zabezpieczyć drzwi po opuszczeniu pomieszczenia.

Urządzenie, które jest efektem końcowym tej pracy, wprowadza dodatkowe zabezpieczenie drzwi do istniejącego rozwiązania mechanicznego. Pozwala na zastąpienie klucza otwierającego mechaniczny zamek kartą lub brelokiem z tagiem RFID. Urządzenie jednak nie wyklucza użycia klucza – w żaden sposób nie blokuje zamka mechanicznego. Jedynym warunkiem jest posiadanie zamka z nakładką po stronie wewnętrznej. Przymocowanie serwomechanizmu do zamka z włożonym kluczem może zablokować możliwość otwarcia drzwi kluczem z zewnątrz. Urządzenie umożliwia kontrolowanie przepływu osób w pomieszczeniu. Każda osoba uprawniona do uzyskania dostępu posiada swoją kartę z unikalnym numerem identyfikacyjnym. Każdorazowe otwarcie drzwi kartą rejestrowane jest przez układ i zapisywane w dziennikach zdarzeń. Urządzenie działa na zasadzie białej listy – jedynie ustalone przez administratora numery identyfikacyjne mogą

uzyskać dostęp. Próba otwarcie drzwi kartą, która nie jest zapisana na białej liście, również jest zapisywana w dzienniku. Dodawanie nowych kart do białej listy trwa tylko parę sekund. Karty kompatybilne z urządzeniem są stosunkowo tanie, więc dodanie wielu kart nie jest dużym kosztem. Zgubienie karty przez posiadacza również nie jest problemem – zgubioną kartę wystarczy usunąć z białej listy, by ewentualny znalazca nie uzyskał dostępu. Po otwarciu drzwi, wejściu do pomieszczenia i ponownym zamknięciu drzwi, zamek przekręcany jest automatycznie na pozycję zamkniętą, co likwiduje potencjalne zagrożenie bezpieczeństwa w przypadku omyłkowego zostawienia niezablokowanych drzwi. Urządzenie posiada dwa przyciski, które razem z ekranem znajdują się po wewnętrznej stronie drzwi. Jednym z nich można umożliwić dostęp osobie, która nie posiada karty.

Po podłączeniu zasilania, urządzenie włącza się automatycznie. Pierwszą czynnością, jaką wykonuje mikrokontroler, jest uruchomienie procedury inicjalizacji. Jeżeli urządzenie jest podłączone do komputera, inicjowane jest połączenie szeregowe poprzez port USB. Zostanie ono wykorzystane do wyświetlenia szczegółowych informacji o stanie urządzenia i aktualnych działaniach. Jest to przydatne przy analizie działania programu i wyszukiwaniu błędów programistycznych. W wersji produkcyjnej urządzenia, informacje te powinny zostać w większości wyłączone, zostawiając jedynie informacje o krytycznych błędach systemu, by ułatwić diagnostykę przy ewentualnej naprawie. Połączenie to jest opcjonalne, urządzenie może działać bez podłączenia do komputera.

Następnym etapem jest uruchomienie wyświetlacza LCD i wyświetlenie komunikatu o rozpoczęciu procesu inicjalizacji (rys. 22).

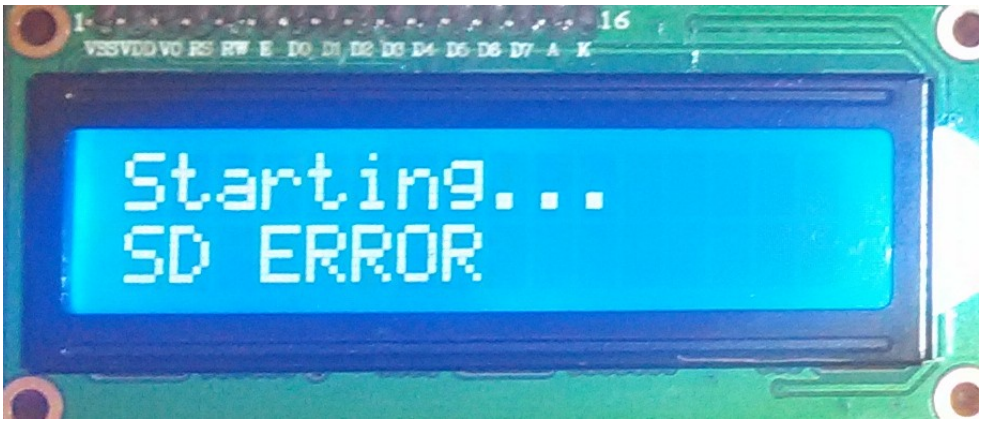
Rysunek 22: Inicjalizacja urządzenia

Uruchomienie ekranu jako pierwszego umożliwi wyświetlenie komunikatów dotyczących kolejnych kroków wykonywanych podczas uruchamiania.

Następnie wybierane są piny dla przycisków i kontaktronu, jednocześnie przypisując przyciskom obsługę przerwań. Uruchomiona zostaje magistrala SPI. Inicjowane są połączenia z pozostałymi modułami, oraz następuje sprawdzenie ich gotowości do działania. Procedura sprawdzająca czytnik kart zbliżeniowych wysyła do niego zapytanie. Jeżeli czytnik nie zwróci odpowiedzi, lub gdy odpowiedź jest nieprawidłowa, wyświetlany jest komunikat o błędzie, a proces inicjalizacji zostaje wstrzymany (rys 23).

Rysunek 23: Błąd czytnika RFID

Podobna procedura sprawdza czytnik kart MicroSD. Po zakończeniu komunikacji z czytnikiem MFCR522, procedura wysyła żądanie do czytnika kart pamięci. Jeżeli nie zwróci odpowiedzi, lub odpowiedź nie będzie poprawna, wyświetlany jest komunikat o błędzie, a inicjalizacja zostaje wstrzymana. Komunikat o błędzie wyświetli się również, gdy do adaptera nie zostanie włożona karta pamięci. Jest ona wymagana do prawidłowej pracy systemu, więc jej brak również przerwie inicjalizację (rys. 24). Powyższe błędy zostaną również wyświetlone w monitorze szeregowym.



Rysunek 24: Błąd czytnika kart microSD

Kolejnym etapem inicjalizacji jest sprawdzenie, czy istnieją już stare dzienniki zdarzeń na karcie pamięci. Urządzenie korzysta z dzienników do zapisywania zdarzeń związanych z jego użytkowaniem. Dzienniki przechowywane są na karcie pamięci MicroSD. Jeżeli istnieje główny dziennik o nazwie „*log.txt”,* zostaje on przeniesiony do pliku o nazwie „*log_oldX.txt”,* gdzie X oznacza kolejny numer dziennika. Jeżeli plik „*log_old0.txt*” już istnieje, dziennik zostanie przeniesiony do pliku z X zwiększonym o 1, aż do osiągnięcia limitu 100 000 plików (rys. 27). Aby sprawdzić pełny dziennik zdarzeń, należy wyłączyć urządzenie i skorzystać z czytnika kart MicroSD w komputerze. Dzięki archiwizacji dziennika, po ponownym włączeniu urządzenie zacznie zapisywać zdarzenia w nowym pliku. Każde wyłączenie urządzenia celem sprawdzenia dziennika spowoduje, że zostanie wygenerowany nowy dziennik. Funkcja ta zapobiegnie spowalnianiu działania urządzenia wraz ze wzrostem rozmiaru pliku dziennika. Zapis i odczyt dużego pliku jest dużo wolniejszy niż mniejszego.

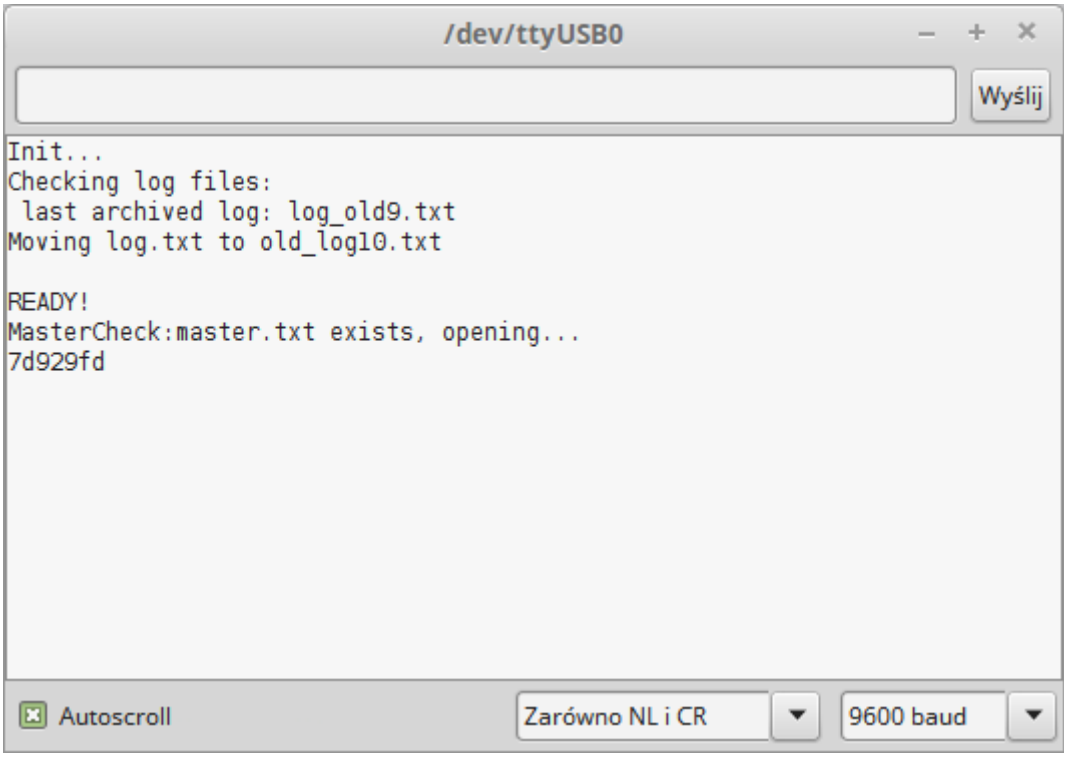
Ostatnim etapem inicjalizacji jest ustawienie serwomechanizmu w pozycji startowej, na której drzwi są zablokowane i zabezpieczone przed nieautoryzowanym dostępem. Po zakończonej inicjalizacji, pierwszą czynnością jest sprawdzenie stanu karty głównej(ang. *Master*). Jeżeli jest to pierwsze uruchomienie, plik zawierający numer głównej karty nie istnieje. Użytkownik zostanie poproszony o wybranie nowej karty głównej (rys. 25).

Rysunek 25: Komunikat o konieczności wybrania głównej karty

Za pomocą tej karty będzie mógł dodawać inne, kompatybilne karty do białej listy. Karta zostanie zapisana na karcie pamięci w pliku „*master.txt*”. Na ekranie LCD zostanie wyświetlony komunikat potwierdzający poprawne zapisanie głównej karty. Cała procedura, razem z inicjalizacją, widoczna jest również w monitorze szeregowym (rys. 26).

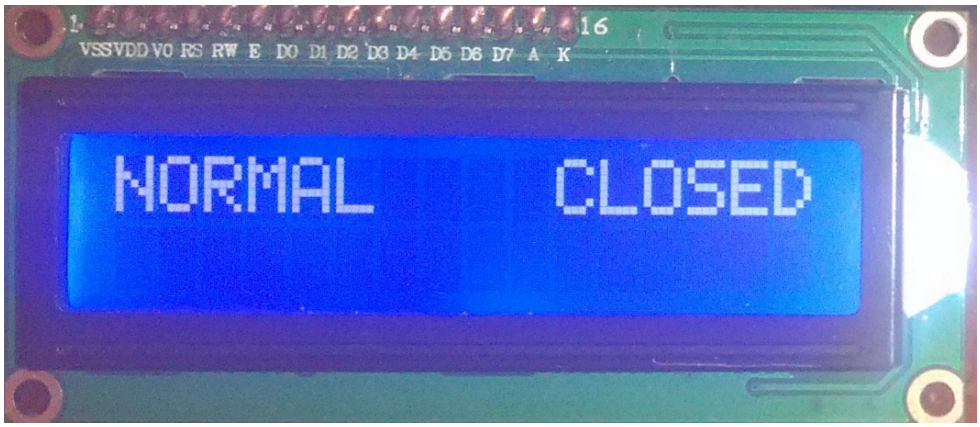
Rysunek 26: Inicjalizacja i dodanie nowej karty głównej

Jeżeli funkcja sprawdzająca wykryje plik „*master.txt*”, odczyta zapisany w nim numer karty master (rys. 27). Odczytany numer zostaje zapisany w pamięci RAM. Plik „*master.txt”* nie będzie otwierany do ponownego uruchomienia urządzenia. Gdy unikalny numer karty głównej znajduje się w pamięci, następuje uruchomienie funkcji oczekującej na nową kartę, czyli przejście do trybu normalnego pracy urządzenia. Istnieją trzy tryby pracy: tryb normalny, tryb historii oraz tryb mastera. W trybie normalnym wysyłany jest sygnał zawierający polecenie żądania dla kart znajdujących się w zasięgu czytnika. W przypadku braku odpowiedzi, następuje kolejna próba, aż do momentu znalezienia karty.



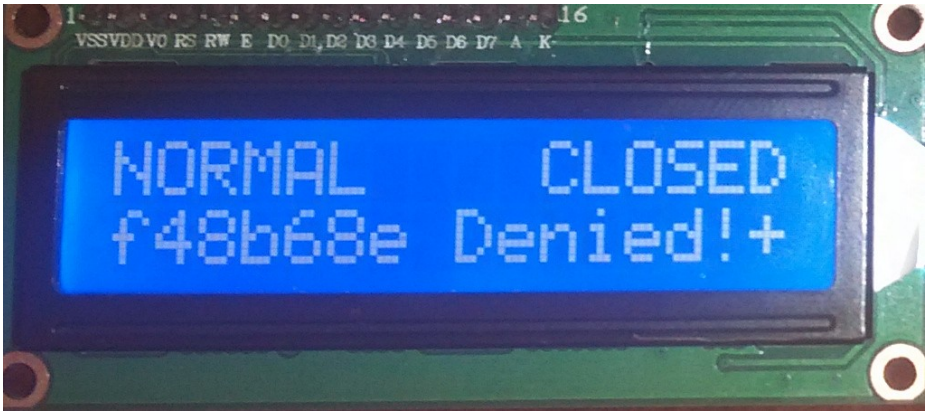
Rysunek 27: Rotacja dziennika, odczytanie UID z pliku master.txt

Zbliżenie nowej karty jest możliwe tylko wtedy, gdy drzwi są zamknięte, ponieważ urządzenie rejestruje każdorazowe odblokowanie drzwi, a nie wejścia do pomieszczenia. Na ekranie wyświetlone zostają dwie informacje: w pierwszym wierszu, z lewej strony, wyświetlana jest informacja o aktualnym trybie pracy urządzenia, a z prawej strony informacja o aktualnym stanie drzwi (rys. 28).



Rysunek 28: Stan po uruchomieniu - tryb normalny, drzwi zamknięte

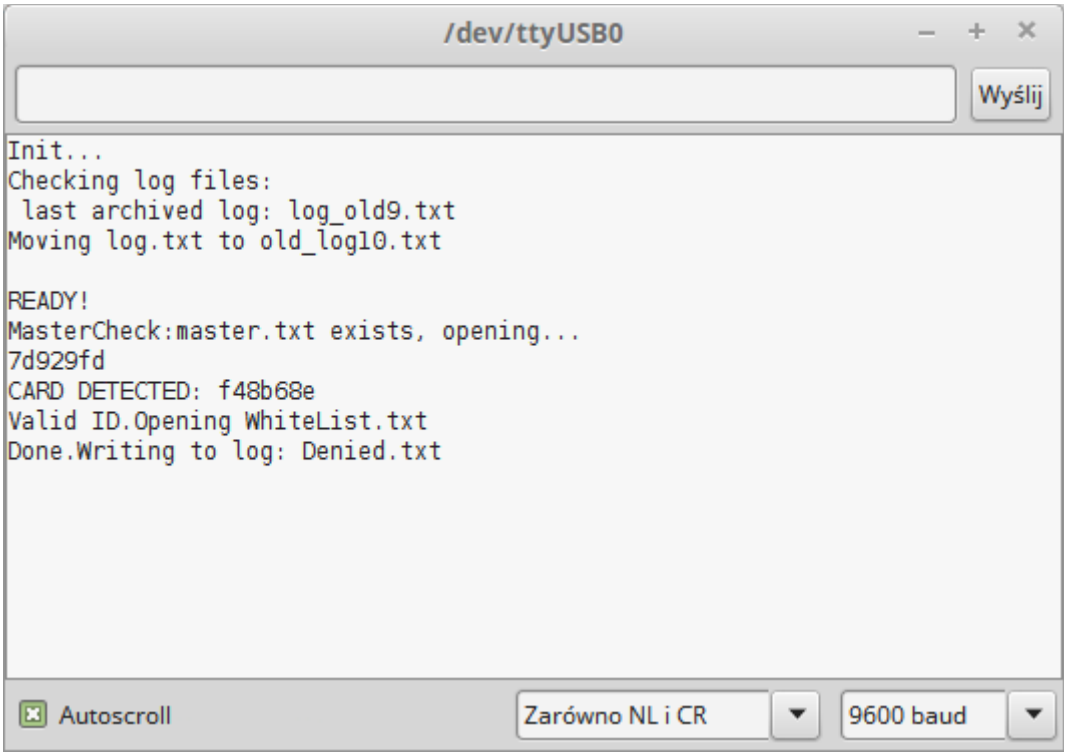
Zbliżenie w tym momencie karty innej niż główna spowoduje wyświetlenie na ekranie numeru tej karty oraz informacji, że karta została odrzucona. Po wykryciu karty system sprawdza, czy odczytany z niej numer ID jest prawidłowy, tj. czy jego długość to dokładnie 7 bajtów (obsługiwane są jedynie karty z numerem o długości 4 bajtów, jednak w programie numery zapisywane są w systemie heksadecymalnym, co daje długość siedmiu bajtów). Kolejnym krokiem jest sprawdzenie, czy karta znajduje się na białej liście. Jeżeli nie, na ekranie zostanie wyświetlony komunikat o braku dostępu (rys. 29).



Rysunek 29: Odmowa dostępu

Próba nieautoryzowanego dostępu zostaje zapisana w dzienniku „Denied.txt”. Znak „+” z prawej strony ekranu oznacza, że zapis na kartę pamięci został prawidłowo wykonany. W przypadku błędu, wyświetlony zostanie znak „-”.

Całe zdarzenie również wyświetlane jest na monitorze szeregowym (rys. 30)

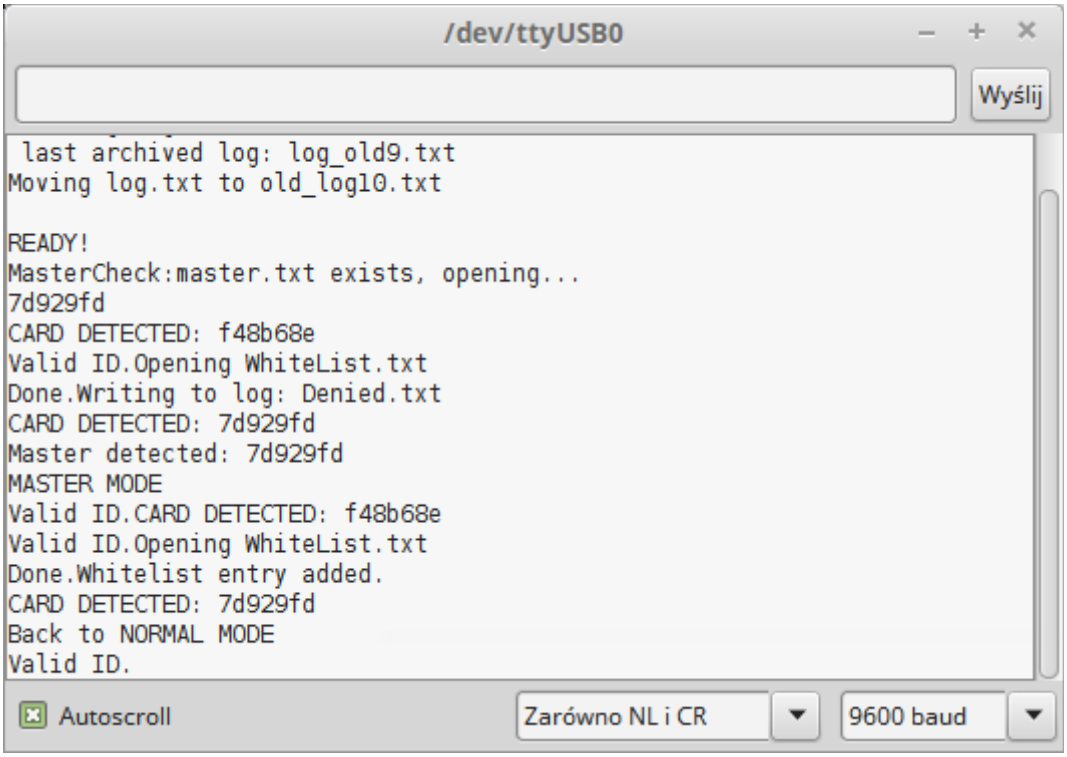


Rysunek 30: Odmowa dostępu

Aby dodać wybraną kartę do białej listy, należy aktywować tryb mastera. Tryb ten włączany jest poprzez zbliżenie karty głównej do czytnika. Przełączenie się trybu sygnalizowane jest poprzez wyświetlenie komunikatu na ekranie LCD. Zbliżenie wcześniej odrzuconej karty spowoduje dodanie jej do białej listy (rys. 31).

Rysunek 31: Dodanie karty do białej listy

Ponowne zbliżenie głównej karty przełączy tryb pracy z powrotem na tryb normalny. Urządzenie czeka na zbliżenie kolejnej karty. Całość widoczna jest na szeregowym monitorze (rys. 32).



Rysunek 32: Dodanie do białej listy i powrót do normalnego trybu

W tym momencie, zbliżenie dodanej do białej listy karty spowoduje przyznanie dostępu. Na ekranie wyświetli się komunikat o odblokowaniu drzwi (rys. 33). Komunikat wyświetla również numer ID karty, która otrzymała dostęp. Będzie on wyświetlany do czasu, aż nastąpi inne zdarzenie - przełączenie trybu pracy lub zbliżenie kolejnej karty. Po odblokowaniu, posiadacz karty ma 7 sekund na uchylenie drzwi. Po tym czasie drzwi zostaną ponownie zablokowane.

Rysunek 33: Przyznanie dostępu

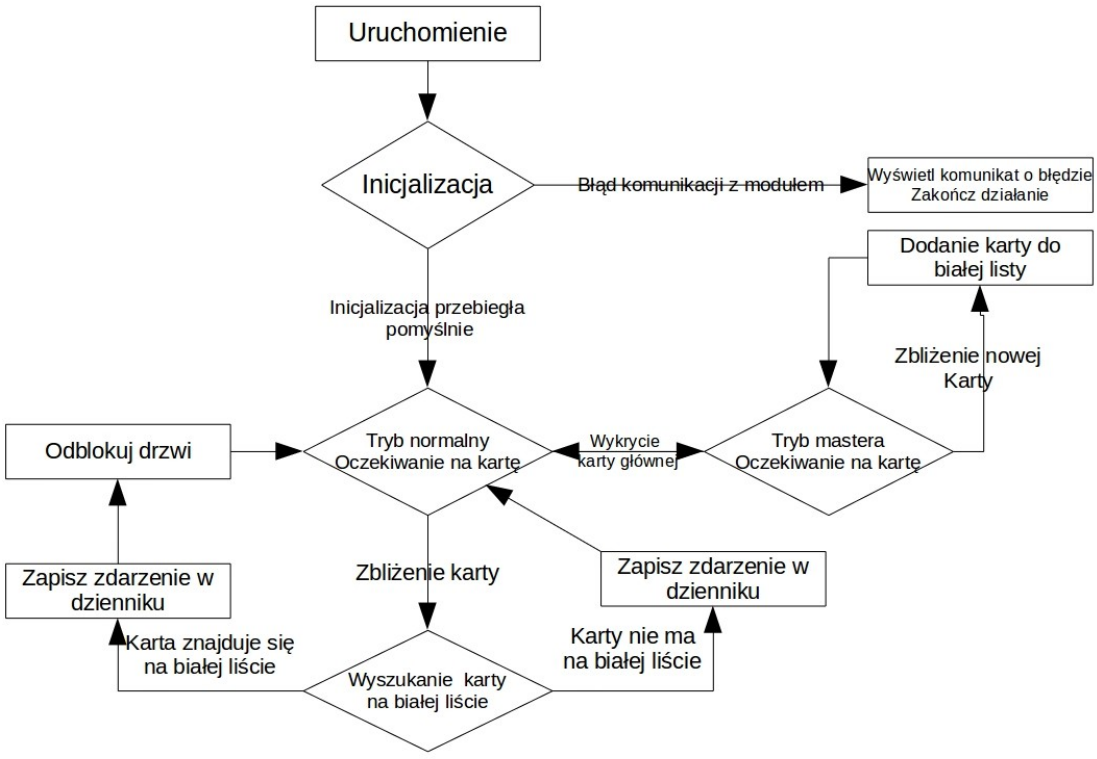
Po uchyleniu drzwi, serwomechanizm nie przekręci zamka, aż do momentu zamknięcia drzwi. Stan, w jakim są aktualnie drzwi kontrolowany jest przez kontaktron. Magnes znajdujący się na framudze zamknie obwód w kontaktronie, gdy drzwi zostaną zamknięte, a urządzenie przekręci zamek, blokując dostęp.

Trzecim trybem pracy jest tryb historii. Wciśnięcie lewego przycisku spowoduje przełączenie urządzenia z trybu normalnego do trybu historii. Warto zaznaczyć, że zbliżenie karty podczas aktywnego trybu historii, automatycznie przełączy urządzenie na tryb normalny. Tryb historii pozwala wyświetlić ostatnie 50 wpisów z dziennika bezpośrednio na ekranie urządzenia. W drugim wierszu ekranu zostaje wyświetlona informacja zawierająca unikalny numer karty oraz czas, który minął od danego wydarzenia (rys. 34).

Rysunek 34: Tryb historii, ostatnia aktywność

Wpisy przechowywane są na karcie w formacie „[numer],[czas od uruchomienia]”. Użytkownik może przełączać kolejne wpisy wykorzystując drugi przycisk. Jeżeli dziennik osiągnie limit 50 wpisów, nastąpi rotacja dziennika, aktualny dziennik zostanie zarchiwizowany i zostanie utworzony nowy.

Arduino Uno R3 nie posiada zegara czasu rzeczywistego, jedynie liczniki, które są w stanie odmierzać czas od momentu uruchomienia mikrokontrolera. Możliwe jest zastosowanie zewnętrznego zegara RTC (ang. *Real Time Clock*), jednak w tym projekcie zostały osiągnięte limity sprzętowe i programowe. Pamięć przeznaczona na program oraz pamięć RAM została prawie całkowicie wykorzystana przez wyżej wymienione funkcje, mimo wielu optymalizacji i modyfikacji kodu, mających na celu zmniejszenie wykorzystanie pamięci. Również gniazda cyfrowe zostały prawie wszystkie wykorzystane, uniemożliwiając podłączenie dodatkowego modułu, jakim jest zegar czasu rzeczywistego. Rysunek 35 przedstawia uproszczony schemat blokowy, który prezentuje kolejne etapy pracy urządzenia z perspektywy zwykłego użytkownika.



Rysunek 35: Uproszczony schemat działania systemu

6.2 Projekt układu elektronicznego

Do realizacji projektu wykorzystano następujące elementy:

- Arduino Uno R3,
- Czytnik RFID MFRC522,
- Adatper dla kart MicroSD Catalex,
- bufor trójstanowy SN74HC125,
- ekran 1602 LCD z adapterem I^2C,
- dwa przyciski,
- dwa rezystory 10k Ω,
- serwomechanizm TowerPro 90g,
- kontaktron Satel K-1

Czytnik kart zbliżeniowych MFRC522 oraz adapter kart MicroSD korzystają z interfejsu szeregowego SPI. Obydwa układy do komunikacji poprzez SPI wykorzystują napięcie 5V, więc nie było potrzeby zastosowania regulatorów napięcia (niektóre moduły korzystają z napięcia 3.3 V do komunikacji interfejsem SPI). Zostały one podłączone do Arduino poprzez wspólne linie MISO, MOSI i SCK, które znajdują się na gniazdach ICSP oraz na pinach cyfrowych 11 (MOSI), 12(MISO) i 13 (SCK). Każdy z tych modułów posiada również oddzielną linię SS (nazwana CS w przypadku karty MicroSD). MFRC522 korzysta dodatkowo z linii RST i SDA do komunikacji z mikrokontrolerem. Posiada również złącze IRQ, które jest wykorzystywane do obsługi przerwań, ale nie jest wspierane przez bibliotekę kompatybilną z Arduino. Na podstawie kodu źródłowego bibliotek stwierdzono, że MFRC522 i adapter kart zaczynają przesuwanie bitów na magistrali SPI od najbardziej znaczącego bitu i korzystają z trybu *SPI_MODE0.*

Podczas projektowania urządzenia, w fazie testów poszczególnych modułów, każde z osobna działało prawidłowo. Jednak po podłączeniu ich razem do magistrali SPI, czytnik kart zbliżeniowych przestawał działać w określonych sytuacjach. Podłączenie było prawidłowe – kod wykorzystany do testowania pojedynczych modułów nadal działał. Próba połączenia kodów dwóch programów powodowała problemy z jednym z modułów. Inicjalizacja czytnika

kart zbliżeniowych jako pierwszego, blokowała działanie adaptera kart MicroSD inicjalizowanego jako drugi w kolejności. W przypadku inicjalizacji urządzeń w odwrotnym porządku, problem pozostawał.

Pierwszą próbą rozwiązania tego problemu było wykorzystanie programowej implementacji SPI dla jednego z modułów. Rozwiązanie to pozwala na wykorzystanie dowolnego pinu cyfrowego do transmisji sygnału do urządzenia korzystającego z SPI. Dzięki temu można zastosować oddzielną magistralę dla każdego urządzenia. Jednak jest to tylko obejście problemu. SPI jest zaprojektowane w taki sposób by mogło z niego korzystać wiele urządzeń. Obsługa programowego SPI w bibliotekach MFRC522 nie była zaimplementowana w trakcie pisania pracy. Wykorzystywana domyślnie biblioteka dla kart MicroSD również nie umożliwia korzystania z programowego SPI. Z tego powodu została wykorzystana najnowsza wersja biblioteki SdFat Williama Greimana, która tę możliwość udostępnia. Przepisanie obsługi kart MicroSD z wykorzystaniem najnowszej biblioteki pozwalającej na wykorzystanie programowe SPI rozwiązało problem. Obydwa moduły mogły działać jednocześnie, gdyż korzystały z oddzielnych magistrali. Jednak zostały wykorzystane aż trzy dodatkowe piny cyfrowe Arduino Uno R3. Wykorzystanie pinów 11,12 i 13 nie jest możliwe, gdy wykorzystane są złącza ICSP do komunikacji poprzez SPI. Gniazda 11,12 i 13 są połączone z ICSP w Arduino Uno R3, więc nie można wykorzystać ich do innych celów[25]. Z tego powodu programowe SPI nie może zostać wykorzystane, ponieważ ilość wolnych gniazd dla pozostałych modułów byłaby niewystarczająca.

Drugą próbą rozwiązania problemu było sprawdzenie, które urządzenie zakłóca transmisję i w jaki sposób. Każde urządzenie podłączone do SPI musi otrzymać sygnał logiczny niski na linii SS, by zacząć wysyłanie danych. Jeżeli urządzenie nie zostało wybrane poprzez ten sygnał, powinno ono odłączyć swoją linię MISO, by nie zakłócać transmisji innych urządzeń. Wystarczyło sprawdzić, które z urządzeń nie rozłącza linii MISO. Źródłem problemu okazał się adapter kart MicroSD. Po inicjalizacji nieprzerwanie nadawał poprzez linię MISO. Aby efektywnie odłączyć ten moduł, zastosowano bufor trójstanowy. Linia CS adaptera jest podłączona jednocześnie do wejścia sterującego bufora trójstanowego oraz do adaptera kart. W tabeli 3 i 4 zaprezentowano sposób podłączenia adaptera kart microSD i czytnika RFID do Arduino Uno R3.

25 ICSP and SPI, [8] http://forum.arduino.cc/index.php?topic=161241.0 [dostęp 11.02.2016]

Tabela 3: Połączenie adaptera z Arduino	
Adapter MicroSD Catalex	**Arduino Uno R3**
CS	Pin D8 (oraz OE bufora)
SCK	ICSP SCK
MOSI	ICSP MOSI
MISO	ICSP MISO (poprzez wej. A i wyj. Y bufora)
VCC	5 V
GND	GND

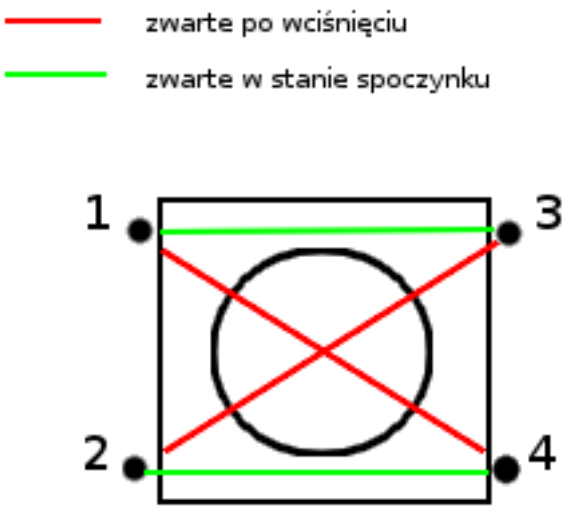
Tabela 4: Połączenie czytnika RFID z Arduino	
MFRC522	**Arduino Uno R3**
SDA	Pin D6
SCK	ICSP SCK
MOSI	ICSP MOSI
MISO	ICSP MISO
IRQ	brak
GND	GND
RST	Pin D5
3.3 V	3.3 V

Na podstawie dokumentacji stwierdzono, że natężenie prądu podczas normalnej pracy (oczekiwanie na kartę – ciągłe wysyłanie sygnału) MFRC522 dochodzi maksymalnie do 100mA. Napięcie robocze czytnika wynosi 3.3 V, czyli maksymalna energia jaką pobiera ten moduł to 0.33 W. Na podstawie pomiarów multimetrem stwierdzono, że adapter kart microSD pobiera maksymalnie do 40mA podczas odczytu i zapisy danych. Podczas czuwania, natężenie prądu dla adaptera wynosi około 25mA. Z tego wynika, że w trybie czuwania adapter pobiera 0.125 W energii, a podczas operacji do 0.2 W.

Serwomechanizm podłączany jest trzema przewodami. Zasilany jest napięciem 5 V. Przewód sterujący podłączono do pinu D9 Arduino. Pin ten ma możliwość generowania sygnału PWM, który wymagany jest do sterowania serwomechanizmem. Na podstawie

pomiarów multimetrem stwierdzono, że natężenie dla serwomechanizmu podczas pracy bez obciążenia to około 150 mA. W zależności od obciążenia, natężenie sięgało do 500mA. Zablokowanie wału serwomechanizmu spowodowało wzrost natężenia do około 700mA przy próbie obrotu. Serwo zasilane jest napięciem 5 V, co przy natężeniu prądu 400mA powoduje pobór energii na poziomie 2 W. W oczekiwaniu na sygnał, natężenie jest bardzo niskie, niemożliwe do zmierzenia standardowym multimetrem.

Do kontrolowania urządzenia wykorzystano dwa przyciski, które podłączono do pinów D2 i D3, obsługujących przerwania. Są to przyciski posiadające cztery nóżki (rys 34).



Rysunek 34: Połączenia w przycisku

Nóżkę 1 w każdym przycisku podłączono do napięcia 5 V. Nożka 4 podłączona jest do wejść Arduino Uno R3 obsługujących przerwania. Nóżka 2 jest podłączona poprzez rezystor podciągający w dół (ang. *pull-down*) o rezystancji 10k Ω. W ten sposób, gdy przycisk znajduje się w stanie spoczynku (nie jest wciśnięty), pin D2 i D3 nie są w stanie wysokiej impedancji. Stan wysokiej impedancji to jest inaczej stan logiczny nieokreślony[26]. W tym stanie wejście losowo przyjmuje albo stan niski, albo stan wysoki – wejście reaguje na zakłócenia występujące w układzie, może także zadziałać jak antena i zmieniać stan reagując

26 [5] Michael Margolis, Arduino Cookbook, O'REILLY (2012), rozdział 5.1 Using a Switch

na zakłócenia zewnętrzne. Pin o nieokreślonym stanie może przejmować również stan pinu sąsiedniego poprzez sprzężenie pojemnościowe. Podłączone w ten sposób przyciski do wejść cyfrowych w Arduino powodują, że w stanie spoczynku na wejściu cyfrowym jest stan logiczny niski, a przytrzymanie przycisku powoduje pojawienie się na wejściu stanu logicznego wysokiego.

Do wyświetlania komunikatów z urządzenia, wykorzystano ekran LCD 1602, połączony z adapterem I^2C. Sposób podłączenia przestawiono w tabeli 5.

Tabela 5: Połączenie LCD 1602

LCD 1602	**Arduino Uno R3**
VCC	5V
GND	GND
SDA	A4
SCL	A5

Ekran jest podświetlany białą diodą LED. Adapter I^2C posiada dodatkowe dwu pinowe złącze, domyślnie z założoną zworką. Można to złącze wykorzystać do sprzętowego kontrolowania podświetleniem ekranu. Na podstawie pomiarów multimetrem stwierdzono, że natężenie prądu dla ekranu LCD i adaptera I^2C wynosi około 25 mA, co daje pobór mocy 0.125 W.

Pierwszy prototyp został przygotowany na płytce stykowej, którego wizualizacja widoczna jest na rysunku 35. Arduino Uno może być zasilane poprzez złącze USB lub zewnętrznym zasilaczem. Standardowo, napięcie podłącza się najpierw do Arduino, a pozostałe moduły zasilane są poprzez gniazda 5 V lub 3.3 V, które znajdują się bezpośrednio na płytce Arduino. Jednakże maksymalne wyjściowe natężenie prądu dla Arduino Uno R3 to 200 mA (dla wyjścia o napięciu 5 V). 200 mA jest już natężeniem, które powoduje zwiększenie się możliwości uszkodzenia układu i może doprowadzić do niestabilnej pracy urządzenia. Przekroczenie tej wartości może spowodować uszkodzenie mikrokontrolera. Maksymalne natężenie dla gniazda 3.3 V to 50 mA, a dla pozostałych pinów to 20 mA.

Zasilanie zewnętrzne można zrealizować wykorzystując zasilacz prądu stałego lub baterię. Zasilacz może zostać podłączony poprzez gniazdo zasilające okrągłe 2.1 mm lub piny

VIN i GND. Płytka może działać na zewnętrznym napięciu w zakresie od 6 V do 20 V. Jednak poniżej 7 V, pin 5V może dostarczać napięcie niższe niż pięć woltów. Podczas korzystania z napięcia większego niż 12 V, regulator napięcia może generować duże ilości ciepła, uszkadzając tym samym cały układ. Zalecany zakres jest od 7 V do 12 V. W projekcie zastosowano zasilanie zewnętrzne podłączone do pinów VIN i GND, ponieważ natężenie prądu na całym układzie jest większe od 200 mA. Na rysunku 23 wspólne przewody podłączone do złącz ICSP na płytce Uno i modułów korzystających z interfejsu SPI zostały wyróżnione osobnymi kolorami:

- różowy – linia MISO,
- brązowy – linia MOSI,
- szary – linia SCK

Kolorem czerwonym i niebieskim zostały oznaczone przewody zasilające układ i poszczególne moduły. Kolory pozostałych przewodów są dobrane przypadkowo (wyjątkiem są przewody serwomechanizmu, które w większości posiadają taką samą kolorystykę).

Najważniejszym elementem pracy jest mikrokontroler, który znajduje się na płytce Arduino UNO R3. Wykorzystując multimetr, zmierzono natężenie prądu dla samej płytki z wgranym programem, który nie obsługuje przełączania mikrokontrolera w tryb uśpienia. Natężenie podczas pracy wyniosło około 50mA, co równa się poborowi 0.25 W energii. W tabeli 6 przedstawiono przybliżone zużycie energii przez urządzenie w trybie czuwania, a w nawiasach zużycie chwilowe, podczas otwierania drzwi.

Tabela 6: Przybliżone parametry całego urządzenia

Moduł	**Napięcie**	**Natężenie (maks)**	**Moc (maks)**
MFRC522	3.3 V	100 mA	330 mW
Adapter MicroSD	5 V	25 mA (40 mA)	125 mW (200 mW)
Serwo SG90	5 V	0 mA (400 mA)	0 mW (2000 mW)
LCD 1602	5 V	25 mA	125 mW
Arduino Uno R3	5 V	50 mA	250 mW
	Łącznie	***200 mA (615 mA)***	***830 mW (2905 mW)***

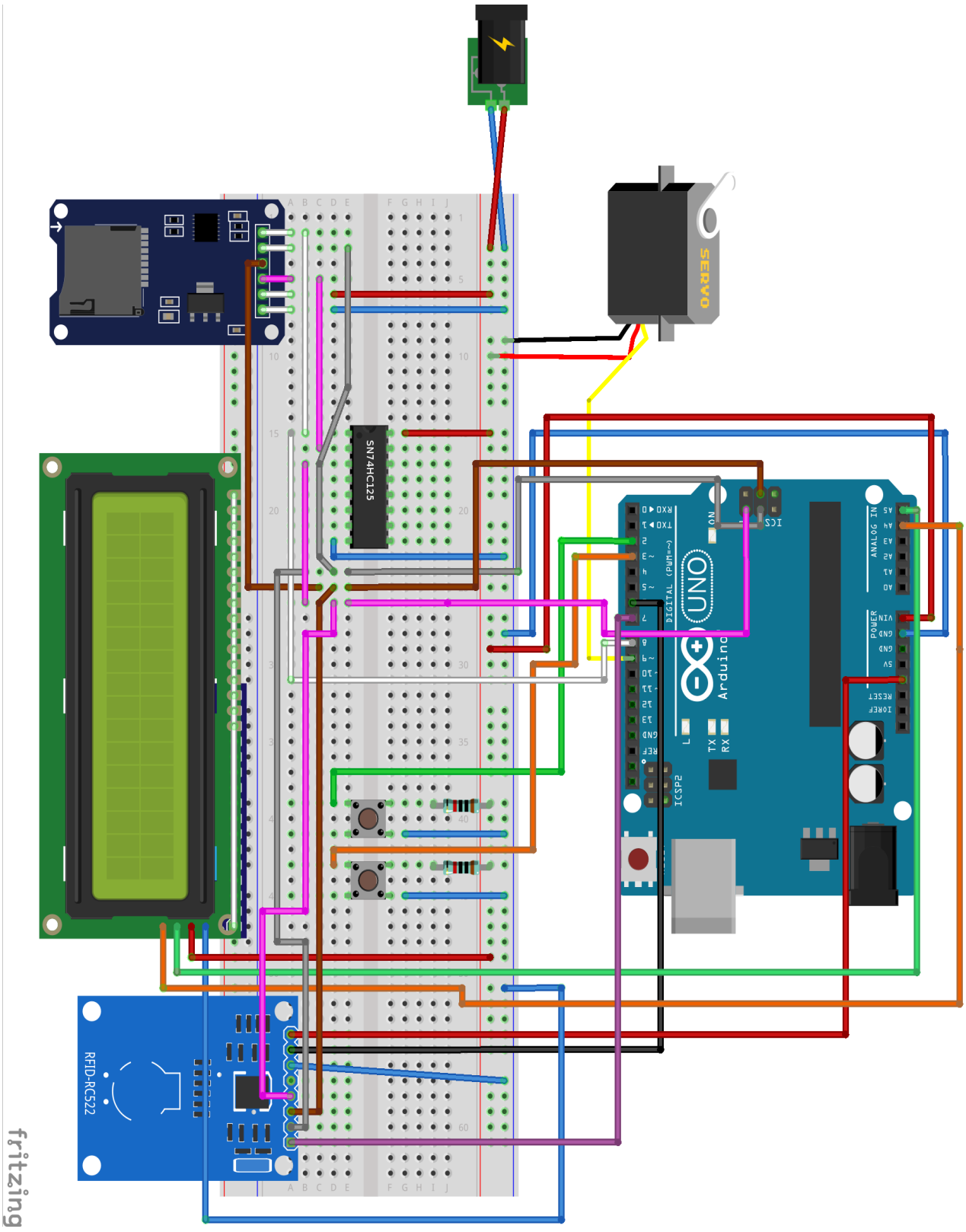
Urządzenie musi być cały czas podłączone do zasilania. Serwomechanizm nie zablokuje zamka, gdy prąd zostanie odcięty. Jednak urządzenie można zabezpieczyć przed utratą napięcia poprzez zastosowanie baterii. Jedną z możliwości byłoby zastosowanie ogniw litowo-polimerowych z dodatkowym układem ładującym te ogniwa. Podczas normalnej pracy ogniwa byłyby ładowane, a w przypadku odcięcia zasilania, energia byłaby dostarczana przez nie dostarczana.

Można wyliczyć przybliżony czas działania na baterii w trybie czuwania. Dla akumulatorów stosowanych w urządzeniach power-bank o pojemności 4400 mAh i napięciu 3.6 V, zmagazynowana energia to 15840 mWh. Rozładowując akumulatory, dla napięcia wyjściowego 5 V, przybliżona pojemność to 3168 mAh. Po uwzględnieniu sprawności układu na poziomie 90%, zostaje około 2851 mAh.
Czas pracy można obliczyć ze wzoru:

$$h=(Ah*V)/W$$

gdzie:
h – czas pracy w godzinach,
Ah – pojemność akumulatora,
V – napięcie akumulatora,
W – moc pobierana przez urządzenie.

Po podstawieniu danych z powyższych obliczeń i z tabeli 6 , uzyskujemy około 19 godzin pracy w trybie czuwania. Czas ten jest wystarczający dla krótkich i średnich przerw w dostawie energii elektrycznej. Czas ten będzie krótszy w zależności od intensywności korzystania z funkcji otwierania drzwi.



Rysunek 35: Schemat prototypu gotowego urządzenia, wykonany w darmowym programie Fritzing

6.3 Budowa kodu źródłowego

Program został napisany w Arduino IDE. Język programowania w nim wykorzystany jest oparty na językach C i C++. Arduino automatycznie generuje prototypy funkcji i przekazuje kod do kompilatora „*avr-g++”*. Arduino obsługuje większość konstruktorów funkcji języka C/C++. Dodaje również swoje, ściślej związane z programowaniem mikrokontrolerów. Zawiera również zestaw bibliotek wykorzystywanych przy programowaniu dodatkowych modułów podłączanych do mikrokontrolera. Poniżej zostaną opisane najważniejsze fragmenty kodu programu. W tym rozdziale z większości funkcji zostaną usunięte fragmenty kodu odpowiedzialne za wyświetlanie komunikatów oraz mniej istotne funkcje. Cały kod źródłowy dostępny jest w załączniku.

Kod programu zaczyna się od zaimportowania bibliotek, widocznych w Listingu 14.

```
1. #include <SPI.h>
2. #include <SdFat.h>
3. #include <MFRC522.h>
4. #include <Wire.h>
5. #include <LiquidCrystal_I2C.h>
6. #include <Servo.h>
```

Listing 14: Import bibliotek

„*SPI.h*” jest biblioteką wykorzystywaną do obsługi interfejsu SPI. Zapewnia podstawowe metody, które są implementacją założeń magistrali. Pozwala w łatwy sposób skonfigurować połączenie z urządzeniami „*slave”,* wykorzystując Arduino jako urządzenie „*master”*. Nie jest wykorzystywana bezpośrednio w projekcie, ale pozostałe biblioteki korzystają z niej do ustanowienia komunikacji z modułami, które są przez te biblioteki obsługiwane.

„*SdFat.h*” to biblioteka, która umożliwia odczytywanie i zapisywanie danych na kartach SD. Jest to minimalna implementacja systemów plików *FAT16* i *FAT32*. Obsługiwane są standardowe karty *SD* oraz karty o wysokiej pojemności – *SDHC*. *SdFat* korzysta ze zmodyfikowanej implementacji SPI, jak i również z implementacji zawartej w Arduino IDE. Dodatkowo umożliwia korzystanie z biblioteki „*iostream”* do zapisywania i odczytywania plików.

„*MFRC522.h*” odpowiada za komunikację z czytnikiem kart zbliżeniowych. Pozwala na odczyt i zapis danych na kartach *MIFARE* typu A. Czytnik wspiera trzy metody komunikacji, ale biblioteka pozwala jedynie na wykorzystanie interfejsu SPI.

Biblioteka „*Wire.h*” pozwala na komunikację z innymi urządzeniami poprzez interfejs I^2C. W projekcie również nie jest bezpośrednio wykorzystywana, jednak jest ona wymagana przez bibliotekę „*LiquidCrystal_I2C.h*”, która umożliwia wyświetlanie tekstu na ekranie LCD. Metody umożliwiające wyświetlenie tekstu są podobne do wykorzystywanych w monitorze szeregowym. Ostatnia biblioteka pozwala na sterowanie serwomechanizmem. Pozycję wału można ustalić wpisując wymagany kąt – nie trzeba ręcznie generować sygnału PWM.

W Listingu 13, w kolejnych wierszach zdefiniowane są stałe globalne za pomocą derektywy „*#define*”.

```
1. #define SD_CS 8
2. #define RFID_RST  6
3. #define RFID_SS  7
4. #define REED 5
5. #define MAX_HIST 50
```

Listing 15: Definicje stałych

Stałe te przechowują numery pinów, do których są podłączone linie poszczególnych modułów. Kontaktron jest podłączony do linii 5, linie RST i SS czytnika RFID kolejno do szóstego i siódmego pinu, a linia CS do pinu ósmego. „*MAX_HIST*” określa maksymalną liczbę wpisów dziennika zdarzeń, jaką urządzenie może zapamiętać w jednym dzienniku. Zwiększenie tej liczby spowoduje konieczność alokacji większej ilości pamięci RAM w mikrokontrolerze. Po zwiększeniu tej liczby do 100, kompilator informuje o możliwości wystąpienia problemów ze stabilnością pracy urządzenia, związaną z niską ilością wolnej pamięci RAM. Z tego powodu zachowano liczbę pięćdziesięciu wpisów na dziennik. Początkowo wykorzystano metodę dynamicznej alokacji pamięci, by wyświetlić ostatnie pięćdziesiąt wpisów dziennika. Wraz z przyrostem wpisów w dzienniku, ilość wykorzystanej pamięci zwiększa się, aż do osiągnięcia limitu. W tym momencie zajętą pamięć przez adresy starych wpisów można zwolnić. Tablica indeksu historii przechowuje lokalizację danego wpisu w dzienniku, jednak pozycja tej lokalizacji w pliku może mieć różny rozmiar. Z tego

powodu zwolnione fragmenty pamięci również mogą być różnego rozmiaru. Próba alokacji pozycji o większym rozmiarze do mniejszego, wolnego obszaru pamięci może spowodować błąd alokacji i nieprawidłowe działanie programu. Zjawisko dzielenia się wolnej pamięci na fragmenty nazywane jest fragmentacją. Rozwiązaniem problemu byłaby defragmentacja pamięci, czyli połączenie mniejszych wolnych obszarów w jeden większy. Jednak w języku C/C++ defragmentacja pamięci nie jest możliwa, ponieważ wymaga ona przeniesienia zajętego obszaru pamięci w inne miejsce, a to spowoduje zmianę adresu tego obszaru. Jeżeli jakiś fragment kodu korzysta z kopii wskaźnika do tego adresu, po defragmentacji przestanie on działać. Z wyżej wymienionych powodów zdecydowano się na zastosowanie stałego rozmiaru dziennika.

Dalej zadeklarowane są tablice znaków przechowujące numer identyfikujący odczytaną kartę oraz numer karty głównej. Niżej zostały zdefiniowane flagi wykorzystywane głównie w instrukcjach warunkowych, występujące w różnych miejscach w kodzie programu, na Listingu 16. Wszystkie nazwy funkcji i zmiennych są w języku angielskim, by ułatwić odczytanie i zrozumienie kodu innym programistom. Program jest napisany dla systemu wbudowanego o ograniczonej pamięci. Funkcjonalność takiego urządzenia musi być dokładnie przemyślana, a program powinien być możliwie jak najmniejszy. Z tego powodu nie ma przyjętej konwencji nazywania funkcji i zmiennych - nazwy w sposób jednoznaczny określają funkcję danego elementu. Nie jest ich dużo, więc nie jest wymagane stosowanie specjalnego nazewnictwa.

```
1. char cardID_HEX[8];
2. char MasterCard[8];
3.
4. bool MasterMode = false;
5. bool ExitMM = false;
6. bool MasterWrite = false;
7. bool MasterPresent = false;
8. bool Authorized = false;
9. bool Opened = false;
10. bool HistoryMode = false;
```

Listing 16: Definicje tablicy znaków i flag

Zmienna tablicowa „*cardID_HEX[8]*” przechowuje odczytaną przez czytnik kartę w systemie heksadecymalnym. „*MasterCard[8]*” przechowuje numer karty głównej odczytany z pliku lub

dodanej po pierwszym uruchomieniu. Pierwsza zmienna „*MasterMode*" przechowuje wartość 1, gdy aktywny jest tryb „*master"*. *„ExitMM"* informuje o stanie pośrednim pomiędzy trybem „*mastera"*,a trybem normalnym, czyli momencie wyjścia z trybu „master*"*. W trakcie przejścia do trybu normalnego, program wykryłby odczytaną kartę główna jako zwykłą kartę, ale dzięki zmiennej „*ExitMM"* można zapobiec wykonaniu niepotrzebnych w tym momencie funkcji. Zmiana kolejności wykonywania funkcji nie jest możliwa, ponieważ karta główna musi być wykryta na samym początku, stąd wykorzystanie dodatkowej zmiennej.

Istnieją trzy główne tryby pracy urządzenia: tryb „*master"*, tryb historii oraz tryb normalny. Jednak, omawiając szczegóły zasady działania urządzenia, można wyodrębnić dodatkowe tryby, takie jak wcześniej opisany tryb pośredni. Kolejnym jest „*MasterWrite"*. Jeżeli numer karty głównej nie znajduje się w pliku „*master.txt*", urządzenie czeka na zbliżenie nowej karty, by zapisać ją jako kartę główną. Ten tryb jest aktywny tylko raz, podczas pierwszego uruchomienia, lub po usunięciu zawartości karty pamięci.

Zmienna „*MasterPresent*" informuje program, czy istnieje zapisana w pamięci karta główna. Służy ona do aktywacji trybów szczegółowych. „Authorized" jest prawdziwe, gdy odczytana zwykła karta znajduje się na białej liście. Po odczytaniu numeru karty i otwarcia drzwi, zmienna ta przywracana jest do pierwotnej wartości, a numer odczytanej karty jest usuwany z pamięci. „*Opened"* przechowuje informacje o jednym z dwóch stanów zamka w drzwiach: odblokowanym lub zablokowanym. Ostatnia zmienna „*HistoryMode*" informuje, czy aktywny jest tryb historii.

W Listingu 17 znajdują się deklaracje zmiennych i jednej stałej odpowiedzialnych za przechowywanie danych, które związane są z obsługą przycisków. „*Volatile*" w deklaracji zmiennej oznacza, że jest to zmienna ulotna, której stan może ulec zmianie pomiędzy dostępami do tej zmiennej. Zmiana ta może zostać wygenerowana przez przerwanie, które jest obsługiwane niezależnie od wykonywanego aktualnie programu. „*Volatile*" informuje również kompilator, by nie optymalizował kodu, w którym znajduje się zmienna poprzedzona tym słowem kluczowym[27].

27 cv (const and volatile) type qualifiers, [4] http://en.cppreference.com/w/cpp/language/cv [dostęp 06.02.2016]

```
1. const byte button[] = {2, 3};
2. volatile bool button_reading[] = {LOW, LOW};
3. volatile unsigned long current_high[2];
4. volatile unsigned long current_low[2];
```

Listing 17: Przechowywanie stanu przycisków

Tablica „*button[]*” przechowuje informacje o pinach, do których podłączone są przyciski. Tablica „button_reading[]” przechowuje informacje o aktualnym stanie przycisków.
Stan niski na wejściu oznacza, że przycisk nie jest aktualnie wciśnięty. W zmiennych tablicowych „*current_high[2]*” i „*current_low[2]*” zapisywany jest czas w milisekundach, w którym nastąpiła zmiana na dany stan przycisku.

Listing 18 zawiera deklaracje pozostałych zmiennych globalnych wykorzystywanych w programie.

```
1. byte log_nr = 0;
2. int idHist[MAX_HIST];
3. int nLine = 0;
4. long openTime;
5. int autoClose = 5000;
```

Listing 18: Pozostałe zmienne

Zmienna „*log_nr*” zawiera numer aktualnie wyświetlanego wpisu w dzienniku. Tablica „*idHist[MAX_HIST]*”, zawiera pozycje wpisów w pliku dziennika, a jej rozmiar jest określony przez stałą „*MAX_HIST*”. Zmienna „*nLine*” przechowuje aktualną liczbę wpisów dziennika, a wykorzystywana jest przy generowaniu indeksu pozycji wpisów w pliku. Zmienna „*openTime*” zawiera czas otwarcia drzwi w milisekundach, a „*autoClose*” określa czas, po którym drzwi zostaną automatycznie zamknięte, jeżeli obwód w kontaktronie będzie zamknięty.

Przed inicjalizacją tworzone są jeszcze uchwyty do modułów podłączonych do płytki Arduino, widoczne na Listing 19. Uchwyt umożliwi korzystanie z metod zawartych w klasie konstruktora, na podstawie którego jest definiowany.

```
1. SdFat MicroSD;
2. SdFile SD_File;
3. MFRC522 Rfid(RFID_SS, RFID_RST);
4. LiquidCrystal_I2C lcd (0x27, 16, 2);
5. Servo servo;
```

Listing 19: Utworzenie instancji modułów

Uchwyt „*SD_File*” pozwoli na korzystanie z metod zawartych w klasie „*SdFat”*, np. *„SD_File.open(„log.txt”)”* spowoduje otwarcie pliku, do zawartości którego można uzyskać dostęp poprzez *„SD_File.read()”*. W przypadku czytnika RFID, należy jeszcze określić numery pinów, do których są podłączone linie SS i RST, a dla ekranu LCD należy ustalić adres I^2C oraz rozmiar wyświetlanego obrazu (liczba znaków na wiersz i liczba wierszy).

Następnie wywołana zostaje funkcja *„setup()”*, która wykonywana jest tylko raz, podczas uruchamiania programu. Inicjalizacja zaczyna się od ustabilizowania połączenia z monitorem szeregowym na komputerze PC i uruchomienia ekranu LCD, co pokazano na Listingu 20.

```
1. void setup()
2. {
3.     Serial.begin(9600);
4.     Serial.println(F("Init..."));
5.     lcd.init ();
6.     lcd.backlight ();
7.     lcd.setCursor(0, 0);
8.     lcd.print(F("Starting..."));
```

Listing 20: Początek inicjalizacji

Wyświetlając tekst na ekranie należy pamiętać o pilnowaniu kursora i czyszczeniu ekranu. Każdy wyświetlony znak na ekranie pozostanie do czasu, aż zostanie on zastąpiony innym znakiem. Należy ręcznie zarządzać kursorem i za każdym razem ustawiać go na żądaną pozycję, wykorzystując metodę *„setCursor(x,y)”*.

Kolejnym etapem inicjalizacji jest przygotowanie przycisków do pracy, poprzez dodanie obsługi przerwań, co pokazano na Listingu 21.

```
1.  pinMode(button[0], INPUT);
2.   pinMode(button[1], INPUT);
3.   pinMode(REED, INPUT);
4.   digitalWrite(REED, HIGH);
5.   attachInterrupt(digitalPinToInterrupt(button[0]), readButton, RISING);
6.   attachInterrupt(digitalPinToInterrupt(button[1]), readButton, RISING);
```

Listing 21: Obsługa przerwań

Aby móc odczytać stan przycisku, należy skonfigurować pin, do którego jest on podłączony jako wejście (ang. *input*). Jeden przewód kontaktronu (ang. *reed*) jest podłączony do pinu piątego, a drugi do masy. W listingu 20 można zauważyć niestandardowe wykorzystanie funkcji *„digitalWrite()”,* która wykorzystywana jest do określania stanu wyjścia. Jednak w tym przypadku dotyczy ona wejścia, ale nie ustala na nim żadnego stanu, tylko uruchamia wbudowany w układ rezystor podciągający *„pull-up*”. Jest to ważne ze względu na to, że gdy obwód w kontaktronie nie jest zamknięty przez magnes, wejście będzie miało stan nieokreślony. Po zastosowaniu tej funkcji, gdy magnes nie zamknie obwodu, wejście będzie w stanie logicznym wysokim. Następnie przypisywana jest obsługa przerwań do funkcji *„readButton()”*, która będzie aktywowana na zboczu rosnącym, czyli po wciśnięciu przycisku. Funkcja widoczna jest na Listingu 22.

```
1. void readButton()
2. {
3.     for (byte i = 0; i < 2; i++)
4.     {
5.           if(digitalRead(button[i]) == HIGH)
6.       {
7.               current_high[i] = millis();
8.               button_reading[i] = HIGH;
9.           }
10.                 if(digitalRead(button[i] == LOW) && button_reading[i] ==
    HIGH)
11.         {
12.             current_low[i] = millis();
13.             If((current_low[i] - current_high[i] > 100 && (current_low[i] -
    current_high[i]) < 800))
14.       {
15.                 button_reading[i] = LOW;
16.           }
17.         }
18.   }
19.}
```

Listing 22: Funkcja „readButton()”

Funkcja ta w pętli sprawdza stan wszystkich przycisków. Zapisuje czas wykrycia wciśnięcia przycisku oraz sprawdza, czy przycisk był wciśnięty dłużej niż 100 milisekund. Celem takiego działania jest filtrowanie zakłóceń, które mogłyby zostać odczytane jako wciśnięcie przycisku.

Dalej uruchomiony zostaje czytnik RFID. Funkcja „*RfidCheck()*” sprawdza, czy połączenie z czytnikiem zostało poprawnie nawiązane. Jeżeli funkcja zwróci wartość równą 0, wyświetlany jest komunikat o błędzie. Przed rozpoczęciem jakiejkolwiek komunikacji z modułem korzystającym z magistrali SPI, należy aktywować go poprzez zmianę stanu jego linii SS lub CS na stan logiczny niski, a pozostałych modułów na stan logiczny wysoki. W ten sposób komunikacja nie zostanie zakłócona przez inne moduły. Do tego celu można wykorzystać funkcję „*digitalWrite()*”, widoczną na listingu 23.

```
1. pinMode(SD_CS, OUTPUT);
2. digitalWrite(SD_CS, HIGH);
3. pinMode(RFID_SS, OUTPUT);
4. digitalWrite(RFID_SS, LOW);
5.
6. SPI.begin();
7. Rfid.PCD_Init();
8. if (!RfidCheck())
9. {
10.         lcd.setCursor(0, 1);
11.         lcd.print("RFID ERROR");
12.         Serial.println(F("RFID Error, check connection!"));
13.         InfiniteLoop()
14.}
```

Listing 23: Inicjalizacja komunikacji z MFRC522

W celu sprawdzenia poprawności inicjalizacji czytnika RFID, wywołana zostaje funkcja „*RfidCheck()”*, pokazaną na Listingu 24. Jej zadaniem jest wysłanie odpowiedniego żądania do czytnika, by ten odpowiedział danymi zawierającymi numer wersji oprogramowania. Jeżeli funkcja żądająca zwróci wartość *0x00* lub *0xFF,* oznacza to, że komunikacja z modułem nie powiodła się, a wtedy funkcja „*RfidCheck()”* zwróci wartość równą 0. Zostanie wyświetlony komunikat o będzie krytycznym i wywołana zostanie funkcja „*infiniteLoop()*”, która jest nieskończoną pętlą, której nie da się zakończyć. Zablokuje to dalsze działanie urządzenia.

```
1. bool RfidCheck()
2. {
3.     byte v = Rfid.PCD_ReadRegister(Rfid.VersionReg);
4.     If ((v == 0x00) || (v == 0xFF))
5.         return false;
       else
6.         return true;
7. }
```

Listing 24: Funkcja "RfidCheck()"

Po poprawnej inicjalizacji czytnika RFID, to samo działanie zostaje wykonane dla adaptera kart MicroSD, pokazanym na Listingu 25). W konfiguracji został wykorzystany parametr

ustalający prędkość transmisji interfejsu SPI na połowę maksymalnej prędkości, zwiększając tym stabilność pracy i zmniejszając ryzyko wystąpienia błędów w transmisji. Maksymalna prędkość nie jest wskazana, ponieważ zapis i odczyt z dziennika zdarzeń transmituje niewielką ilość danych. Jeżeli inicjalizacja przebiegła pomyślnie, uruchamiana jest funkcja „*rotateLog()*”, która sprawdza, czy na karcie istnieje już plik o nazwie „*log.txt*”. Jeżeli tak, to zawartość zostaje przeniesiona do archiwum (szerzej opisane w rozdziale 6.1).

```
1. if (!MicroSD.begin(SD_CS, SPI_HALF_SPEED)
2. {
3.     lcd.setCursor(0, 1);
4.     lcd.print(F("SD ERROR"));
5.     Serial.println(F("SD Error, check if card is inserted."));
6.     infiniteLoop();
7. }
8. rotateLog();
9. digitalWrite(SD_CS, HIGH);
10. digitalWrite(RFID_SS, LOW);
11. servo.attach(9);
12. servo.write(0);
13. delay(600);
14. servo.detach();
```

Listing 25: Inicjalizacja czytnika kart MicroSD oraz ustawienie serwomechanizmu

W pliku „*log.txt*” może znajdować się maksymalnie 50 wpisów. Gdy limit zostanie osiągnięty, zawartość przenoszona jest do innego pliku. Również po ponownym uruchomieniu urządzenia dziennik jest przenoszony do archiwum. Do tego celu wykorzystywana jest funkcja „*rotateLog()*” (listing 26). Jej działanie zaczyna się od sprawdzenia, czy plik dziennika już istnieje. Jeżeli tak, to pętla szuka wolnej nazwy dla pliku archiwum – funkcja „*sprintf()*” wstawia w miejsce znacznika „*%s*” wartość zmiennej „*i*”, która jest inkrementowana w każdym przebiegu pętli. Nowy dziennik zostanie utworzony w momencie wywołania funkcji zapisującej zarejestrowanie nowej karty. Maksymalna liczba plików jest równa liczbie stu tysięcy. Gdy limit zostanie osiągnięty, wyświetli się komunikat ostrzegający o zapełnieniu pamięci. Nowe pliki nie zostaną utworzone.

```
1. void rotateLog() {
2.     if (MicroSD.exists("log.txt"))
3.     {
4.         Serial.println(F("Checking log files:"));
5.         char old_log_name[8];
6.         for (int i = 0; i < 100000; i++)
7.         {
8.             sprintf(old_log_name, "log_old%d.txt", i);
9.             if (!MicroSD.exists(old_log_name))
10.                 {
11.                     MicroSD.rename("log.txt", old_log_name);
12.                     if (i == 100000)
13.                     {
14.                         lcd.setCursor(0, 1);
15.                         lcd.print(F("SD FULL"));
16.                         Serial.println(F("LOG LIMIT REACHED!"));
17.                     }
18.                  break;
19.                 }
20.             }
21.         }
22.     }
```

Listing 26: Funkcja "rotateLog()"

Po zakończeniu operacji na dziennikach, wał serwomechanizm zostaje ustawiony na pozycję początkową poprzez metodę „*write(x)*”, gdzie x jest równy wybranemu kątowi. Metoda „*detach()*” wyłącza serwo. Bez wywołania tej metody, serwomechanizm postara się utrzymać wał na ustawionej wcześniej pozycji, co skutkowałoby zablokowaniem zamka. Na końcu procesu inicjalizacji zostają jeszcze wyświetlone odpowiednie komunikaty na ekranie LCD i monitorze szeregowym.

W głównej pętli programu zawarty jest najważniejsza część kodu. Na początku tej pętli znajduje się kod odpowiedzialny za obsługę przycisków, widoczny na Listingu 27.

```
1.  void loop()
2.  {
3.      for (int i = 0; i < 2; i++)
4.      {
5.          if (button_reading[1] == HIGH)
6.          {
7.              if (HistoryMode && !MasterMode)
8.              {
9.                  log_nr--;
10.                 showLogEntry(log_nr);
11.                 if (log_nr == 0)
12.                     log_nr = nLine + 1;
13.                 }
14.                 if (!HistoryMode)
15.                     zamienDrzwi();
16.                 button_reading[1] = LOW;
17.             }
18.             if (button_reading[0] == HIGH)
19.             {
20.                 if (!HistoryMode)
21.                 {
22.                     Serial.println(F("HISTORY"));
23.                     HistoryMode = true;
24.                     log_nr = nLine;
25.                     showLogEntry(log_nr);
26.                     lcdClearLine(0, 6, 0);
27.                     lcd.setCursor(0, 0);
28.                     lcd.print(F("HISTORY"));
29.                 }
30.                 else
31.                 {
32.                     exitHistory();
33.                 }
34.                 button_reading[0] = LOW;
35.             }
36.         }
```

Listing 27: Obsługa przycisków

Instrukcja warunkowa sprawdza wartość zmiennej tablicowej „*button_reading[]*”, by określić, czy został wciśnięty przycisk. Instrukcja wykonywana jest w pętli dla wszystkich przycisków. Pierwszy przycisk, jeżeli aktywny jest tryb normalny, wywołuje funkcję „*zamienDrzwi()*”, która zamyka lub otwiera drzwi w zależności od wartości zmiennej

„*Opened*”. Jeżeli aktywny jest tryb historii, przycisk ten dekrementuje zmienną „*log_nr*”, co skutkuje wyświetleniem kolejnych wpisów z dziennika zdarzeń. Drugi przycisk zmienia tryb pracy urządzenia z normalnego na tryb historii i odwrotnie, wykorzystując flagę „*HistoryMode*”. Za wyświetlanie wpisów odpowiada funkcja „*showLogEntry()*”, widoczna na Listingu 28.

```
1.  void showLogEntry(byte log_nr)
2.  {
3.      if (SD_File.open("log.txt", O_READ))
4.      {
5.          char c;
6.          char time[10];
7.          int time_num = 0;
8.          int i = 0;
9.           SD_File.seekSet(idHist[log_nr] - 7);
10.                 for (int i = 0; i < 7; i++)
11.                 {
12.                     c = SD_File.read();
13.                     lcd.print(c);
14.                 }
15.                 SD_File.seekCur(1);
16.                 c = SD_File.read();
17.                 while (c > 47 && c < 58)
18.                 {
19.                     time[i] = c;
20.                     i++;
21.                     c = SD_File.read();
22.                 }
23.                 time_num = atoi(time);
24.                 //------tutaj znajduje się kod wyświetlający czas w odpowiednim
    formacie
25.             }
26.          SD_File.close();
27.          }
```

Listing 28: Funkcja "showLogEntry()"

Parametrem funkcji jest pozycja wpisu w dzienniku, który ma zostać wyświetlony użytkownikowi. Metoda „*seekSet()*” ustawia kursor w pliku na wybraną pozycję. Jako parametr przyjmuje pozycję przecinka w wybranym wpisie, która przechowywana jest w tablicy „*IdHist[]*”. Każdy wpis przechowywany jest w formacie „numerID,czas”. Przecinek jest dobrym punktem odniesienia, który można wykorzystać do wyszukiwania wybranego

wpisu, ponieważ występuje zawsze po siedmiu znakach numeru ID karty. Czas zapisany po drugiej stronie przecinka może mieć różną długość, ale na końcu zawsze znajduje się znak końca linii. Z tego powodu czas jest odczytywany w pętli „*while*” (wiersz 17), która wykonuje się, gdy odczytany bajt jest dziesiętnym kodem „*ASCII*” w zakresie od 47 do 58. Zakres ten odpowiada literom alfabetu oraz cyfrom. Odczytany czas jest zapisany w tablicy znaków. Aby przeprowadzić na nim obliczenia, można wykorzystać funkcję „atoi()”, która przeprowadzi konwersję tablicy znaków na zmienną liczbowej „*integer*”. Pozycje przecinków we wpisach dziennika, które przechowywane są w tablicy „*IdHist[]*”, generowane są w funkcji „*genIndex()*”, która wywoływana jest podczas zapisu nowego zdarzenia na karcie pamięci. Funkcja pokazana jest na Listingu 29.

```
void genIndex()
{
    if (SD_File.open("log.txt", O_READ))
    {
        int nByte = 0;
        char cByte;
         nLine = -1;
        while (nByte = SD_File.available())
        {
            cByte = SD_File.read();
           nByte++;
           if (cByte == ',')
            {
                 nLine++;
                idHist[nLine] = nByte - 1;
            }
        }
    }
SD_File.sync();
SD_File.close();
}
```

Listing 29: Funkcja "genIndex()"

Funkcja odczytuje plik bajt po bajcie, zapisując pozycję każdego napotkanego przecinka. Metoda „*available()*” zwraca ilość pozostałych znaków do końca pliku. Metoda „*seekSet()*” przyjmuje jako parametr pozycję liczoną od początku pliku i w ten sposób zapisywana jest lokalizacja przecinka w tablicy „*IdHist*”.

Przed rozpoczęciem wykrywania kart zbliżeniowych, należy najpierw sprawdzić, czy w systemie zapisany jest numer identyfikacyjny karty głównej. Działanie to wykonywane jest przez funkcję „*MasterRead*”, pokazaną na Listingu 30.

```
1.  void MasterRead()
2.  {
3.     if (SD_File.open("master.txt", O_READ))
4.     {
5.        for (int i = 0; i < 7; i++)
6.         {
7.             MasterCard[i] = SD_File.read();
8.         }
9.         SD_File.close();
10.              MasterPresent = true;
11.           }
12.           else
13.         {
14.               MasterWrite = true;
15.           }
16.           return;
17.       }
```

Listing 30: Funkcja "MasterRead()"

Odczytuje ona unikalny identyfikator karty głównej z pliku „*master.txt*”. Jeżeli plik nie istnieje, ustawia wartość zmiennej „MasterWrite” jako 1. Następny odczytany numer ID będzie zapisany jako karta główna. Do odczytania numeru ID karty wykorzystywana jest funkcja „*readTagId()*” , która wywoływana jest, gdy zostanie spełniony jednen warunek – drzwi muszą być zamknięte. Funkcja znajduje się w Listingu 31. Pierwszą czynnością w funkcji jest wykorzystanie metody wykrywającej karty znajdujące się w zasięgu czytnika RFID. Brak karty przerwie działanie całej funkcji, która będzie za chwile wywołania ponownie, a cykl będzie się powtarzał aż do wykrycia karty, by następnie odczytać jej numer identyfikacyjny. Czterobajtowy numer zostaje zapisany w systemie heksadecymalnym.

```
1.  bool readTagID()
2.  {
3.          if ( ! Rfid.PICC_IsNewCardPresent())
4.                  return false;
5.          if ( ! Rfid.PICC_ReadCardSerial())
6.                  return true;
7.          if (HistoryMode)
8.                  exitHistory();
9.          byte cardID[4];
10.                 String tempID;
11.                 for (int i = 0; i < 4; i++)
12.                 {
13.                         cardID[i] = Rfid.uid.uidByte[i];
14.                         tempID +=  String(cardID[i], HEX);
15.                 }
16.                 tempID.toCharArray(cardID_HEX, 8);
17.                 Rfid.PICC_HaltA();
18.                 return false;
19.         }
```

Listing 31: Funkcja "readTagId()"

Za zapis na karcie odpowiada funkcja „*Write2SD()*”, w której rodzaj zapisanych danych zależny jest od aktualnie aktywnych flag, widocznych na Listingu 32. Jako parametr przyjmuje numer karty, który ma zostać zapisany na karcie pamięci. Jeżeli aktywnym trybem jest „*MasterWrite*”, to celem zapisu jest plik „*master.txt*”. Jeżeli aktywny jest tryb „*Master*”, numer zostaje zapisany na białej liście. Przed zapisem numeru ID na białą listę wywołana zostaje funkcja „*look4id()*”, która sprawdza, czy numer nie został już wcześniej zapisany. Jeżeli żaden z powyższych trybów nie jest aktywny, numer karty zostaje zapisany w dwóch dziennikach zdarzeń. Jednym z i nich jest dziennik ogólny, a drugi jest przypisany do numeru ID karty. Dla każdej karty tworzony jest osobny dziennik.

```
void Write2SD(char FcardID[8])
{
	if (MasterWrite)
	{
		if (SD_File.open("master.txt", O_RDWR | O_CREAT | O_AT_END))
		{
			SD_File.println(FcardID);
			SD_File.close();
			MasterWrite = false;
			for (int i = 0; i < 8; i++)
			{
				MasterCard[i] = FcardID[i];
			}
			MasterPresent = true;
		}
	}
	else if (MasterMode && (strcmp(MasterCard, FcardID) != 0))
	{
		if (!look4id("WhiteList.txt", FcardID))
		{
		if (SD_File.open("WhiteList.txt", O_RDWR | O_CREAT |
O_AT_END))
		{
			SD_File.println(FcardID);
			SD_File.close();
		}
	}
	else
		lcd.print(F(" EXISTS"));
	}
	else if (!MasterMode && !ExitMM)
	{
		if (Authorized)
	{
		SaveLog(FcardID, FcardID, 0);
		SaveLog(FcardID, "log", 1);
	}
	else
	{
		SaveLog(FcardID, "Denied", 1);
	}
	if (nLine == MAX_HIST - 1)
		rotateLog();
	}
}
```

Listing 32: Funkcja "Write2SD()"

Szkic używa 27512 bajtów z (85%) pamięci programu. Maksimum to 32256 bajtów. Globalne zmienne używają 1451 bajtów z (70%) dynamicznej pamięci, pozostawiając 597 bajtów dla lokalnych zmiennych. Maksimum to 2048 bajtów. W związku z wykorzystaniem dużej części pamięci RAM przez zmienne globalne, sprawdzono ilość wolnej pamięci w różnych miejscach programu. Globalne zmienne przechowywane są na stercie. Kompilator informuje o rozmiarze sterty, która została zaalokowana w pamięci RAM – jej rozmiar jest znany po zakończonej kompilacji. Należy zatem sprawdzić wykorzystanie pamięci RAM przez stos, w którym przechowywane są lokalne zmienne alokowane dynamicznie. Badanie zostało wykonane w kilku kluczowych miejscach programu. Wyniki przedstawione zostały w tabeli 7. Do badania wolnej ilości pamięć RAM, wykorzystana została funkcja „*freeRam()*”, widoczna na Listingu 33, która zwraca ilość wolnej pamięci w bajtach.

```
int freeRam () {
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}
```

Listing 33: Funkcja "freeRam()"

Tabela 7: Ilość wolnej pamięci RAM podczas wykonywania wybranych funkcji

Nazwa funkcji	**Wolna pamięć RAM w bajtach**
rotateLog	537
ReadTagID	561-547
SaveLog	555
look4id	564
showLogEntry	565
genIndex	548

Na podstawie wyników z tabeli 7 można stwierdzić, że program nie wykorzystuje całej dostępnej pamięci RAM, co jest gwarantem stabilnej pracy.

7. Podsumowanie

Projekt zabezpieczenia pomieszczeń przed nieautoryzowanymi osobami został zbudowany w oparciu o platformę Arduino, który jest oparty na wolnym i otwartym oprogramowaniu. Jest on ciągle rozwijany przez społeczność, a jego popularność ciągle rośnie. Rozbudowana dokumentacja i bardzo duża dostępność różnego rodzaju materiałów jest zdecydowanym atutem tej platformy. Pozwala ona na doskonalenie umiejętności programowania i tworzenia układów elektronicznych. Różnorodność modeli płytek Arduino umożliwia ich zastosowanie w wielu dziedzinach życia codziennego. Pozwala na szybkie opracowywanie inteligentnych systemów wbudowanych. Niski koszt zachęca do wypróbowania swoich sił i przetestowania umiejętności i wiedzy. Tworzony przez społeczność bogaty zbiór kursów i projektów pozwala rozwijać posiadaną już wiedzę.

Aby zbudować dobre urządzenie, należy zapoznać się z technologiami, które wykorzystywane są do komunikacji pomiędzy urządzeniami. Zrozumienie zasady działania każdej z nich pozwoli na zaprojektowanie urządzenia w najbardziej optymalny sposób. Pozwoli to również na dobranie modułów, które będą w stanie ze sobą współpracować.

Przy projektowaniu układu elektronicznego ważne jest zastosowanie źródła zasilania, które zapewni wszystkim modułom parametry wymagane do poprawnej i stabilnej pracy. Arduino posiada kilka możliwości podłączenia źródła zasilania, a wybranie odpowiedniego rozwiązania jest bardzo ważnym aspektem projektu. Wykorzystanie serwomechanizmu może znacznie obciążyć mikrokontroler, jeżeli zostanie do niego podłączony bezpośrednio. Razem z innymi modułami, może doprowadzić do uszkodzenia płytki Arduino.

Głównym problemem przy projektowaniu urządzenia były ograniczenia sprzętowe zastosowanego modelu płytki Arduino. Arduino Uno R3 oparte jest o mikrokontroler, którego pamięć jest bardzo ograniczona. Zbudowane na nim urządzenie nie może być zbytnio skomplikowane, jego funkcjonalność musi być dokładnie zaplanowana, by w trakcie tworzenia nie osiągnąć limitów narzuconych przez mikrokontroler. Z tego powodu ważna również jest optymalizacja programu. Algorytmy powinny być możliwie jak najmniejsze, przy jednoczesnym zachowaniu pełnej, zaplanowanej wcześniej funkcjonalności. Należy pamiętać, że pamięć operacyjna jest ograniczona. Typy zmiennych powinny być dobrane w

taki sposób, by zajmowały tylko tyle pamięci, ile naprawdę zostanie wykorzystane w programie.

Zbudowane urządzenie jest dobrym przykładem zastosowania mikrokontrolerów w życiu codziennym. Jest to dobre rozwiązanie dla małych firmy, które chcą wprowadzić dodatkowe zabezpieczenie wybranych pomieszczeń i możliwość kontrolowania korzystających z nich pracowników. Nowe rodzaje zabezpieczeń są ważnym elementem nowoczesnych systemów. Rozwój technologii pozwala na skuteczniejsze łamanie zabezpieczeń, ale jednocześnie pozwala na projektowanie coraz bardziej skutecznych rozwiązań odpornych na ich łamanie. Wraz z rozwojem coraz bardziej wydajniejszych i tańszych metod produkcji elementów elektronicznych, popularne dziś mechaniczne metody zabezpieczenia osób i mienia zostaną zastąpione przez techniki cyfrowe. Z czasem klasyczne klucze zostaną zastąpione ich cyfrowymi odpowiednikami. Platformy takie jak Arduino pozwalają już dziś na zaprojektowanie swojego własnego systemu zabezpieczeń, który można dostosować do indywidualnych potrzeb.

Urządzenie, które zostało opisane w tej pracy zostało zaprojektowane jako samodzielne, przeznaczone do montażu na jednych drzwiach. Projekt może zostać rozbudowany o system, który pozwoli na zdalne zarządzanie wieloma takimi urządzeniami. Urządzenie mogłoby zostać podłączone do internetu, wykorzystując do tego celu moduł Wi-Fi ESP8266. Funkcjonalność trybu „*master”* przejęta została by przez serwer, a przypisywanie kart mogłoby odbywać się w jednym miejscu, niezależnie od ilości urządzeń znajdujących się w sieci. Dzienniki zdarzeń przechowywane w pamięci urządzenia mogłyby w całości zostać przeniesione na serwer. Taki system mógłby zostać wykorzystany w większych przedsiębiorstwach.

8. Literatura

[1] Boxall John (2014), Arduino. 65 praktycznych projektów, Helion.

[2] Deshmukh V Ajay (2005), Microcontrollers: Theory and Applications, Tata McGraw-Hill Education.

[3] Eady Fred (2004), Networking and Internetworking with Microcontrollers, Elsevier.

[4] Igoe Tom (2012) ,Getting Started with RFID: Identify Objects in the Physical World with Arduino, O'REILLY.

[5] Margolis Michael (2012), Arduino Cookbook, O'REILLY.

[6] Wheat Dale (2011), Arduino Internals, Apress.

[7] Wójcik Marcin (2000), hierarchizacja ważności wynalazków wpływających na postęp elektroniki, Akademia Górniczo – Hutnicza, Kraków.

9. Artykuły internetowe

[1] AttachInterrupt(), https://www.arduino.cc/en/Reference/AttachInterrupt [dostęp 09.02.2016]

[2] Arduino FAT16/FAT32 Library, https://github.com/greiman/SdFat [dostęp 06.02.2016]

[3] Arduino UNO & Genuino UNO, https://www.arduino.cc/en/Main/ArduinoBoardUno [dostęp 03.02.2016].

[4] Cv (const and volatile) type qualifiers, http://en.cppreference.com/w/cpp/language/cv [dostęp 06.02.2016]

[5] Dokumentacja techniczna LCD1602, http://www.elecrow.com/download/LCD1602.pdf [dostęp 04.02.2016]

[6] Dokumentacja techniczna PCF8574, http://www.nxp.com/documents/data_sheet/PCF8574.pdf [dostęp 04.02.2016].

[7] Dokumentacja techniczna MFRC522, http://www.nxp.com/documents/data_sheet/MFRC522.pdf [dostęp 20.01.2016].

[8] "How to Make Zuse's Z3 a Universal Computer", http://www.inf.fu-berlin.de/users/rojas/1997/Universal_Computer.pdf [dostęp 11.02.2016].

[9] ICSP and SPI, http://forum.arduino.cc/index.php?topic=161241.0 [dostęp 11.02.2016]

[10] Intel and TI – Microprocessors and Microcontrollers, http://www.datamath.org/Story/Intel.htm [dostęp 08.02.2016]

[11] Introduction to I²C and SPI protocols, http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/?/article/AA-00255/22/Introduction-to-SPI-and-IC-protocols.html [dostęp 02.02.2016].

[12] MFRC52, http://playground.arduino.cc/Learning/MFRC522, [dostęp 02.04.2016]

[13] Serial Peripheral Interface Bus, https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus [dostęp 03.02.2016].

[14] Specyfikacja techniczna TowerPro SG90, http://datasheet.sparkgo.com.br/SG90Servo.pdf [dostęp 10.01.2016].

[15] The Chip that Jack Built, http://www.ti.com/corp/docs/kilbyctr/jackbuilt.shtml [dostęp 28.01.2016].

10. Wykaz rysunków

11. Wykaz listingów

12. Wykaz tabel

13. Załączniki

13.1 Kod źródłowy programu

```
#include <SPI.h>
#include <SdFat.h>
#include <MFRC522.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
#include <Servo.h>

#define SD_CS 8
#define RFID_RST  6
#define RFID_SS  7
#define REED 5
#define MAX_HIST 50
char cardID_HEX[8];
char MasterCard[8];
bool MasterMode = false;
bool ExitMM = false;
bool MasterWrite = false;
bool MasterPresent = false;
bool Authorized = false;
bool Opened = false;
bool HistoryMode = false;

const byte button[] = {2, 3};
volatile bool button_reading[] = {LOW, LOW};
volatile unsigned long current_high[2];
volatile unsigned long current_low[2];

byte log_nr = 0;
int idHist[MAX_HIST];
int nLine = 0;
long openTime;
int autoClose = 5000;

SdFat MicroSD;
SdFile SD_File;
MFRC522 Rfid(RFID_SS, RFID_RST);
LiquidCrystal_I2C lcd (0x27, 16, 2);
Servo servo;

void setup() {
  Serial.begin(9600);
  Serial.println(F("Init..."));
  lcd.init ();
  lcd.backlight ();
```

```
lcd.setCursor(0, 0);
lcd.print(F("Starting..."));

pinMode(button[0], INPUT);
pinMode(button[1], INPUT);
pinMode(REED, INPUT);
digitalWrite(REED, HIGH);
attachInterrupt(digitalPinToInterrupt(button[0]), readButton, RISING);
attachInterrupt(digitalPinToInterrupt(button[1]), readButton, RISING);

pinMode(SD_CS, OUTPUT);
digitalWrite(SD_CS, HIGH);
pinMode(RFID_SS, OUTPUT);
digitalWrite(RFID_SS, LOW);

SPI.begin();

Rfid.PCD_Init();
if (!RfidCheck()) {
  lcd.setCursor(0, 1);
  lcd.print("RFID ERROR");
  Serial.println(F("RFID Error, check connection!"));
  infiniteLoop();
}

digitalWrite(RFID_SS, HIGH);
digitalWrite(SD_CS, LOW);

if (!MicroSD.begin(SD_CS, SPI_HALF_SPEED))
{
  lcd.setCursor(0, 1);
  lcd.print(F("SD ERROR"));
  Serial.println(F("SD Error, check if card is inserted."));
  infiniteLoop();
}
rotateLog();

digitalWrite(SD_CS, HIGH);
digitalWrite(RFID_SS, LOW);

servo.attach(9);
servo.write(0);
delay(600);
servo.detach();

lcdClearLine(0, 15, 0);
lcd.setCursor(0, 0);
lcd.print(F("NORMAL"));
```

```
  lcdClearLine(13, 16, 0); lcd.setCursor(10, 0); lcd.print("CLOSED");
  Serial.println(F(" "));
  Serial.println(F("READY!"));
}

void readButton() {
  for (byte i = 0; i < 2; i++)
  {
    if (digitalRead(button[i]) == HIGH) {
      current_high[i] = millis();
      button_reading[i] = HIGH;
    }
    if (digitalRead(button[i] == LOW) && button_reading[i] == HIGH) {
      current_low[i] = millis();
      if ((current_low[i] - current_high[i] > 100 && (current_low[i] - current_high[i]) < 800)) {
        button_reading[i] = LOW;
      }
    }
  }
}
void loop()
{
  for (int i = 0; i < 2l; i++)
  {
    if (button_reading[1] == HIGH)
    {
      if (HistoryMode && !MasterMode)
      {
        log_nr--;
        showLogEntry(log_nr);
        if (log_nr == 0)
          log_nr = nLine + 1;
      }
      if (!HistoryMode)
        zamienDrzwi();
      button_reading[1] = LOW;
    }
    if (button_reading[0] == HIGH)
    {
      if (!HistoryMode)
      {
        Serial.println(F("HISTORY"));
        HistoryMode = true;
        log_nr = nLine;
        showLogEntry(log_nr);
        lcdClearLine(0, 6, 0);
        lcd.setCursor(0, 0);
        lcd.print(F("HISTORY"));
```

```
    }
    else
    {
      exitHistory();
    }
    button_reading[0] = LOW;
  }

}
Authorized = false;
digitalWrite(RFID_SS, HIGH);
digitalWrite(SD_CS, LOW);

if (!MasterPresent && !MasterWrite)
  MasterRead();

digitalWrite(SD_CS, HIGH);
digitalWrite(RFID_SS, LOW);

if (!Opened)readTagID();

if (strcmp(MasterCard, cardID_HEX) == 0 && MasterPresent && !MasterMode) {
  MasterMode = true;
  Serial.print(F("Master detected: "));
  Serial.println(MasterCard);
  Serial.println(F("MASTER MODE"));

  lcdClearLine(0, 6, 0);
  lcdClearLine(0, 15, 1);
  lcd.setCursor(0, 0);
  lcd.print(F("MASTER"));
}
else if (strcmp(MasterCard, cardID_HEX) == 0 && MasterPresent && MasterMode)
{
  MasterMode = false;
  ExitMM = true;
  lcdClearLine(0, 6, 0);
  lcdClearLine(0, 15, 1);
  lcd.setCursor(0, 0);
  lcd.print(F("NORMAL"));
  Serial.println(F("Back to NORMAL MODE"));
}

if (strlen(cardID_HEX) == 7 && !Opened)
{
  Serial.print(F("Valid ID.")); //db
  if (!MasterMode && !ExitMM)
  {
```

```
      if (look4id("WhiteList.txt", cardID_HEX))
        Authorized = true;
      lcdClearLine(7, 15, 1);
      lcd.setCursor(7, 1);
      if (Authorized)
      {
        lcd.print(F(" Granted"));
        zamienDrzwi();
      }
      else
        lcd.print(F(" Denied!"));
    }
    digitalWrite(RFID_SS, HIGH);
    digitalWrite(SD_CS, LOW);
    Write2SD(cardID_HEX);
    digitalWrite(SD_CS, HIGH);
    digitalWrite(RFID_SS, LOW);
    clearCardID();
  }
  else
    clearCardID();

  if (millis() - openTime > autoClose && Opened) {
    if (!digitalRead(REED))
    {
      Serial.println(F("Auto:"));
      delay(2000);
      zamienDrzwi();
    }
  }
  ExitMM = false;
}

void genIndex() {
  if (SD_File.open("log.txt", O_READ))
  {
    int nByte = 0;
    char cByte;
    nLine = -1;
    while (SD_File.available())
    {
      cByte = SD_File.read();
      nByte++;
      if (cByte == ',')
      {
        nLine++;
        idHist[nLine] = nByte - 1;
```

```
      }

    }
   }
   SD_File.sync();
   SD_File.close();

}
void showLogEntry(byte log_nr)
{
  if (SD_File.open("log.txt", O_READ))
  {
    char c;
    char time[10];
    int time_num = 0;
    int i = 0;
    SD_File.seekSet(idHist[log_nr] - 7);
    Serial.print("ID: ");
    lcdClearLine(0, 15, 1);
    lcd.setCursor(0, 1);
    for (int i = 0; i < 7; i++)
    {
     c = SD_File.read();
     Serial.print(c);
     lcd.print(c);
    }
    lcd.print(F(" "));
    Serial.println();
    Serial.print("TIME: ");
    SD_File.seekCur(1);
    c = SD_File.read();
    while (c > 47 && c < 58)
    {
     time[i] = c;
     i++;
     Serial.print(c);
     c = SD_File.read();
    }
    time_num = atoi(time);
    int now = millis() / 1000;
    if (now - time_num < 60)
    {
     lcd.print(now - time_num);
     lcd.print("s ago");
    }
    else if (now - time_num > 60  && now - time_num < 3600 * 6)
    {
     lcd.print((now - time_num) / 60);
```

```
      lcd.print("m ago");
    }
    else if (now - time_num > 3600 * 6 && now - time_num < 3600 * 96)
    {
      lcd.print((now - time_num) / 3600);
      lcd.print("h ago");
    }
    else if (now - time_num > 3600 * 96)
    {
      lcd.print((now - time_num) / 3600 * 24);
      lcd.print("d ago");
    }
  }
  SD_File.close();
}
bool look4id(char filename[9], char cardID[8])
{
  char idBuffer[8];
  byte chPos = 0;
  bool exist = false;
  idBuffer[7] = '\0';
  if (SD_File.open(filename, O_READ))
  {
    Serial.print(F("Opening "));
    Serial.println(filename);
    while (SD_File.available())
    {
      char ch = SD_File.read();
      if (ch == '\n')
      {
        chPos = 0;
        if (strcmp(idBuffer, cardID) == 0)
        {
          SD_File.close();
          Serial.print(F("Already in file."));
          exist = true;
        }
      }
      else if (chPos < 7)
      {
        idBuffer[chPos] = ch;
        chPos++;
      }
    }
    Serial.print(F("Done."));
    SD_File.close();
  }
  else
```

```
    Serial.print(F("ID doesn't exists! "));
  return exist;
}
bool RfidCheck()
{
  byte v = Rfid.PCD_ReadRegister(Rfid.VersionReg);
  if ((v == 0x00) || (v == 0xFF))
    return false;
  else
    return true;
}
void MasterRead()
{
  Serial.print(F("MasterCheck:"));
  if (SD_File.open("master.txt", O_READ))
  {
    Serial.println(F("master.txt exists, opening..."));
    for (int i = 0; i < 7; i++)
    {
      MasterCard[i] = SD_File.read();
    }
    SD_File.close();
    Serial.println(MasterCard); /
    MasterPresent = true;
  }
  else {
    Serial.println(F("master.txt doesn't exist, waiting for new card..."));
    lcdClearLine(0, 15, 0);
    lcdClearLine(0, 15, 1);
    lcd.setCursor(0, 0);
    lcd.print(F("SET ANY CARD"));
    lcd.setCursor(0, 1);
    lcd.print(F("AS MASTER NOW"));
    MasterWrite = true;
  }
  return;
}
bool readTagID()
{
  if ( ! Rfid.PICC_IsNewCardPresent())
    return false;
  if ( ! Rfid.PICC_ReadCardSerial())
    return true;

  if (HistoryMode)
    exitHistory();
  Serial.print(F("CARD DETECTED: "));
  byte cardID[4];
```

```
  String tempID;
  for (int i = 0; i < 4; i++)
  {
   cardID[i] = Rfid.uid.uidByte[i];
   tempID +=  String(cardID[i], HEX);
  }
  tempID.toCharArray(cardID_HEX, 8);
  lcdClearLine(0, 15, 1);
  lcd.setCursor(0, 1);
  lcd.print(cardID_HEX);
  Serial.println(cardID_HEX);
  Rfid.PICC_HaltA();
  return false;
}
void SaveLog(char FcardID[8], char log_file[9], bool global)
{
  char filename[14];
  sprintf(filename, "%s.txt", log_file);
  if (SD_File.open(filename, O_RDWR | O_CREAT | O_AT_END))
  {
   if (global)
   {
    SD_File.print(FcardID);
    SD_File.print(F(","));
    SD_File.println(millis() / 1000);
   }
   else
   {
    SD_File.println(millis() / 1000);
   }
   SD_File.close();
   SD_File.sync();
   if(log_file == "log")
   genIndex();
   lcd.setCursor(15, 1);
   lcd.print(F("+"));
   Serial.print(F("Writing to log: "));
   Serial.println(filename);
  }
  else
  {
   Serial.print(F("Cannot open: "));
   Serial.println(filename);
   lcd.setCursor(15, 1);
   lcd.print(F("-"));
  }
}
```

```
void Write2SD(char FcardID[8])
{
 if (MasterWrite)
 { //zapis mastera na karte
  if (SD_File.open("master.txt", O_RDWR | O_CREAT | O_AT_END))
  {
   SD_File.println(FcardID);
   SD_File.close();
   Serial.println(F("MasterCard saved."));
   MasterWrite = false;
   lcdClearLine(0, 15, 0);
   lcdClearLine(0, 15, 1);
   lcd.setCursor(0, 0);
   lcd.print(F("MASTER SET"));
   for (int i = 0; i < 8; i++)
   {
    MasterCard[i] = FcardID[i];
   }
   MasterPresent = true;
   //delay(1500);
   lcdClearLine(0, 16, 0);
   lcd.setCursor(0, 0);
   lcd.print(F("NORMAL"));
   lcd.setCursor(10, 0);
   if (!Opened)
    lcd.print(F("CLOSED"));

  }
  else
  {
   lcd.setCursor(15, 1);
   lcd.print(F("-"));
  }
 }
 else if (MasterMode && (strcmp(MasterCard, FcardID) != 0))
 {
  if (!look4id("WhiteList.txt", FcardID))
  {
   if (SD_File.open("WhiteList.txt", O_RDWR | O_CREAT | O_AT_END))
   {
    SD_File.println(FcardID);
    SD_File.close();
    Serial.println(F("Whitelist entry added."));
    lcdClearLine(0, 15, 1);
    lcd.setCursor(0, 1);
    lcd.print(FcardID);
    lcd.print(F(" ADDED"));
   }
```

```
    }
    else
      lcd.print(F(" EXISTS"));
  }
  else if (!MasterMode && !ExitMM)
  {
    if (Authorized)
    {
      SaveLog(FcardID, FcardID, 0);
      SaveLog(FcardID, "log", 1);
    }
    else
    {
      SaveLog(FcardID, "Denied", 1);
    }
    if (nLine == MAX_HIST - 1)
      rotateLog();
  }
}
void lcdClearLine(byte kolumnaOd, byte kolumnaDo, byte wiersz)
{
  for (byte i = kolumnaOd; i <= kolumnaDo; i++) {
    lcd.setCursor(i, wiersz); lcd.print(" ");
  }
}

void infiniteLoop()
{
  Serial.print(F("CRITICAL ERROR!")); //db
  while (true)
  {
    delay(1000);
  }
}
void clearCardID()
{
  for (int i = 0; i <= 7; i++)
  {
    cardID_HEX[i] = 0;
  }
}
void zamienDrzwi()
{
  servo.attach(9);
  if (!Opened)
  {
    servo.write(180);
    lcdClearLine(13, 16, 0); lcd.setCursor(10, 0); lcd.print("OPENED");
```

```
    Opened = true;
    Serial.println(F("Opening doors"));
    openTime = millis();
  }
  else
  {
    servo.write(0);
    lcdClearLine(13, 16, 0); lcd.setCursor(10, 0); lcd.print("CLOSED");
    Opened = false;
    Serial.println(F("Closing doors"));
  }
  delay(600);
  servo.detach();
}
void rotateLog() {
  if (MicroSD.exists("log.txt"))
  {
    Serial.println(F("Checking log files:"));
    char old_log_name[8];
    for (int i = 0; i < 100000; i++)
    {
      sprintf(old_log_name, "log_old%d.txt", i);
      if (!MicroSD.exists(old_log_name))
      {
        Serial.print(F(" last archived log: "));
        Serial.println(old_log_name);
        Serial.print("Moving log.txt to old_log"); Serial.print(i + 1); Serial.println(".txt");
        MicroSD.rename("log.txt", old_log_name);
        if (i == 100000)
        {
          lcd.setCursor(0, 1);
          lcd.print(F("SD FULL"));
          Serial.println(F("LOG LIMIT REACHED!"));
        }
        break;
      }
    }
  }
}
void exitHistory()
{
  HistoryMode = false;
  Serial.println(F("NORMAL MODE"));
  lcdClearLine(0, 6, 0);
  lcdClearLine(0, 15, 1);
  lcd.setCursor(0, 0);
  lcd.print(F("NORMAL"));
}
```