Projekt na Systemy Wbudowane — **DOKUMENTACJA**:

Prosta gra oparta na sterowaniu pojazdem i ostrzeliwaniu przeszkód, z rozgrywką wyświetlaną na ekranie LCD.

Zespół:

- Bartłomiej Paszkowski, 180658, 180658@edu.p.lodz.pl
- Kamil Kozłowski, 189704, 189704@edu.p.lodz.pl
- Jakub Rogalski 189758, 189758@edu.p.lodz.pl

Wykorzystano:

- Komputer **Arduino UNO R3**
- Funkcjonalności MCU:
    - Wszystkie zegary Arduino
    - Przerwania systemowe
    - Pamięć EEPROM.
    - Mechanizm resetu
- Urządzenia peryferyjne
    - Potencjometr (analogowy)
    - Przycisk (cyfrowy)
    - Ekran LCD podłączony za pomocą gniazda VGA
    - Głośnik

Opis projektu

Celem projektu było stworzenie pseudokonsoli do gier, z jedną prostą grą umieszczoną w pamięci. Sterowanie odbywa się za pomocą potencjometru i przycisku. Obraz generowany przez komputer jest wyświetlany na ekranie LCD. Ponadto, w tle odtwarzana jest melodia.

Gra umieszczona w pamięci opiera się na sterowaniu małym myśliwcem. Gracz musi unikać nadlatujących przeszkód oraz niszczyć je przy pomocy wystrzelonych pocisków. Celem gry jest przetrwać jak najdłużej bez kolidowania z przeszkodami.

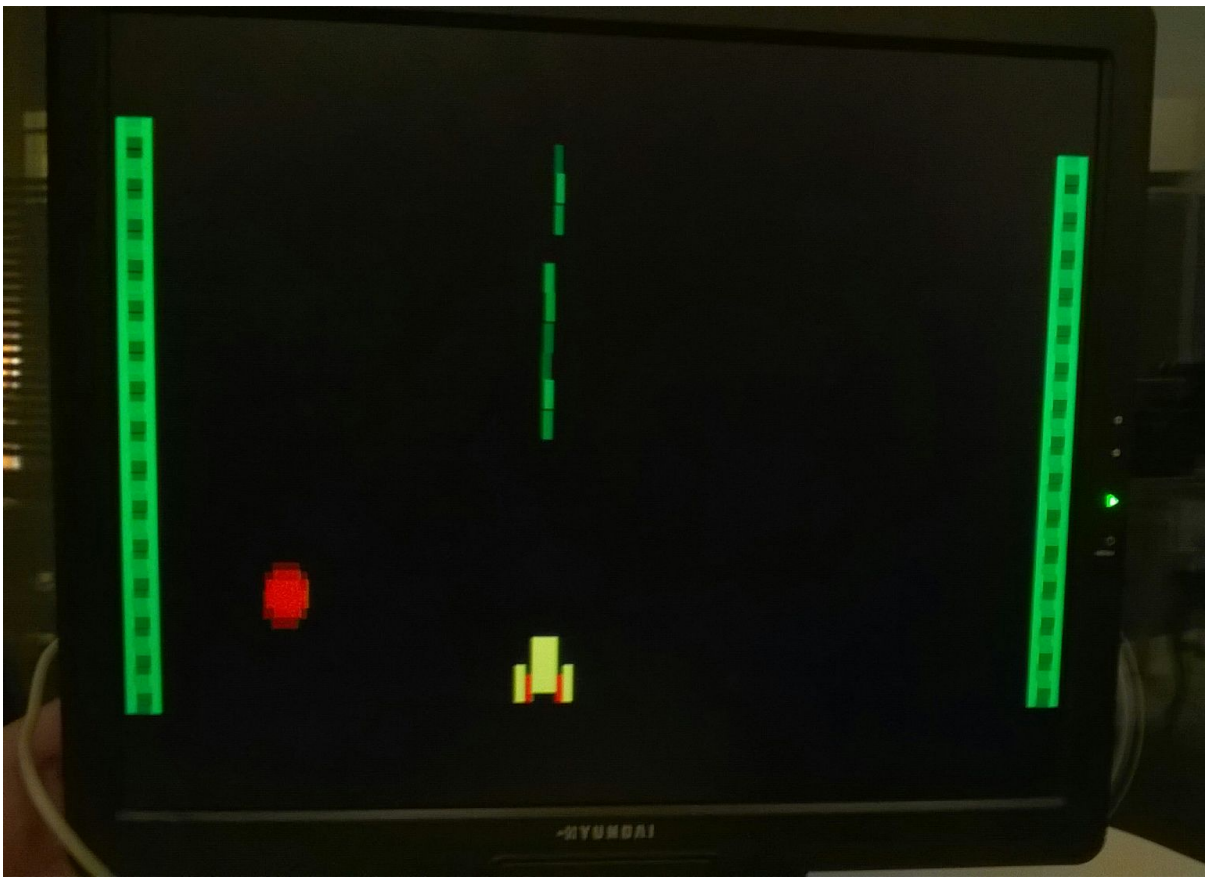
Podział pracy

Praca w zespole rozkładała się bardzo równomiernie.

Bartłomiej Paszkowski	**34%**	udostępnił własny egzemplarz komputera Arduino, peryferia, oraz połączył je ze sobą w układ elektroniczny. Stworzył komunikacje komputera z ekranem LCD. Stworzył ruch rakiety oraz reprezentacje graficzną wszystkich wyświetlanych elementów.

Kamil Kozłowski	**33%**	Zaprogramował odtwarzanie odpowiedniej melodii w odpowiednich sytuacjach. Stworzył też kod ruchomych przeszkód czyhających na gracza, oraz pocisków, które wystrzeliwuje rakieta, poruszanych w głównej pętli programu. Dodał efekty śmierci gracza. Opracował sterowanie za pomocą przycisku i potencjometru, oraz restartowanie przyciskiem.
Jakub Rogalski	**33%**	Zaprogramował zapis danych do pamięci EEPROM, oraz odczyt z niej, implementował obsługę przerwań i timerów. Tworzył system audio, pisał logikę gry, w tym kolizje obiektów.

Sposób obsługi (dokumentacja użytkownika)



Gracz jest reprezentowany jako żółto-czerwony statek kosmiczny w okolicy dolnej krawędzi ekranu. Położeniem statku w poziomie można sterować z a pomocą wychyleń potencjometru.
Przy górnej części ekranu pojawiają się regularnie przeszkody, które w stałym tempie zmierzają ku dołowi. Gracz może zniszczyć przeszkodę, trafiając ją pociskiem. By wystrzelić pocisk, należy nacisnąć czarny przycisk.

W przypadku gdy statek wpadnie na przeszkodę, pojawi się ekran końcowy. By opuścić ekran końcowy i rozpocząć na nowo grę, należy wcisnąć przycisk strzału.

Sposób działania

Wszystkie elementy wyświetlane na ekranie są oddzielnymi obiektami. W głównej pętli programu (`loop()`) są wywoływane funkcje aktualizacji każdego obiektu z osobna. Wewnątrz tej funkcji każdy obiekt aktualizuje swoje położenie, oraz pobiera informacje o sterowaniu (wychylenie potencjometru, wciśnięcie przycisku), a także umieszcza swoją reprezentację graficzną w tablicy kolorów (o rozmiarze 120x60 pól) przesyłanej potem na ekran.

Obiekty wprowadzają zmiany w położeniu raz na określoną liczbę cykli procesora. wykrywanie kolizji jest prowadzone na podstawie wspomnianej wcześniej tablicy kolorów.

Główna funkcja, wykonywana przez Arduino po uruchomieniu, wygląda w kodzie źródłowym biblioteki Arduino (`Arduino/hardware/arduino/avr/cores/arduino/main.cpp`) następująco:

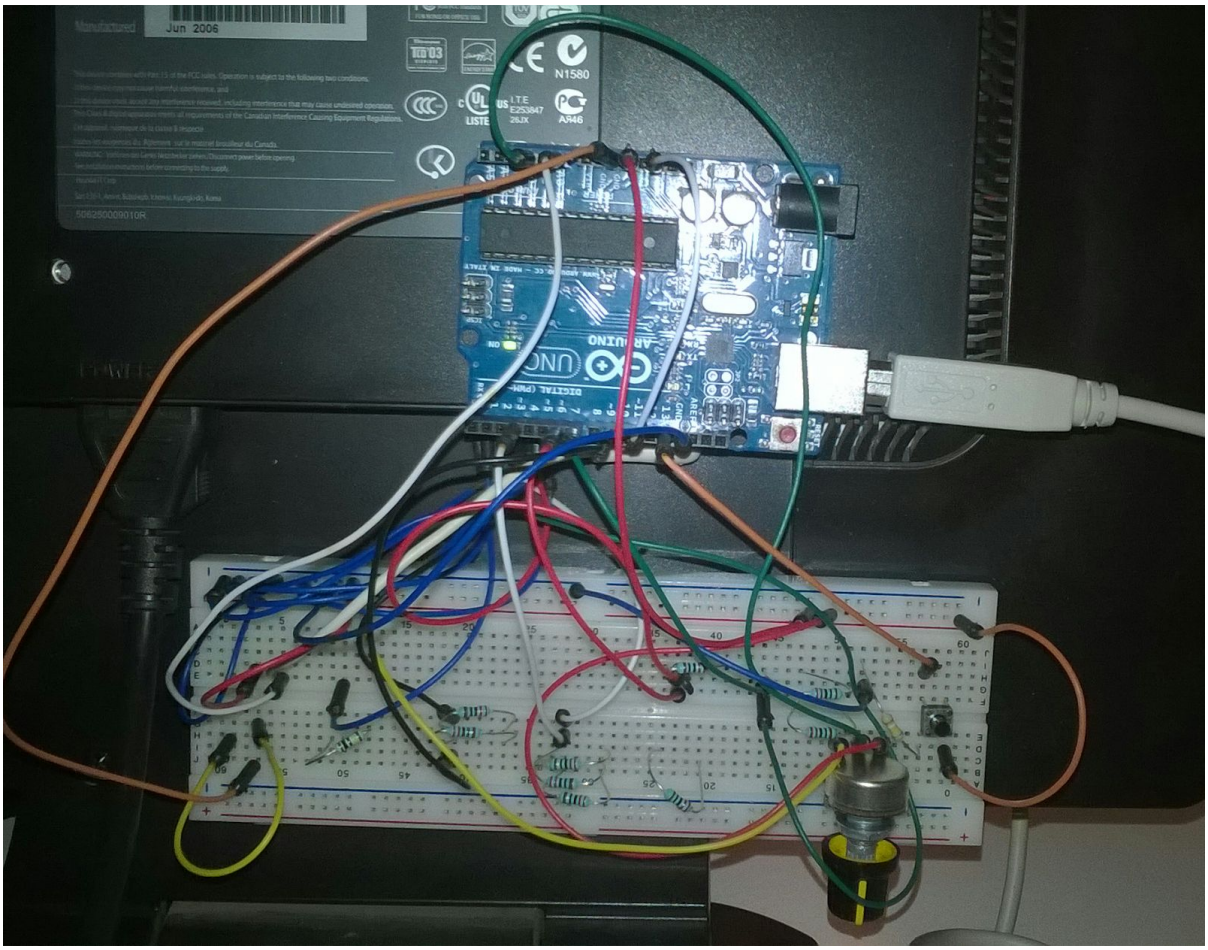
```
int main(void)
{
	init();
	initVariant(); //podstawowe, wewnętrzne funkcje inicjalizujące

	setup(); //funkcja, w której ustawiamy tryby działania pinów i inicjalizujemy
		 //Grę. Mamy do niej dostęp z wewnątrz projektu - w pliku .ino

	for (;;) {
		loop(); //funkcja głównej pętli programu - również mamy do niej dostęp.
	}

	return 0;
}
```

Funkcjonalności



Ekran LCD

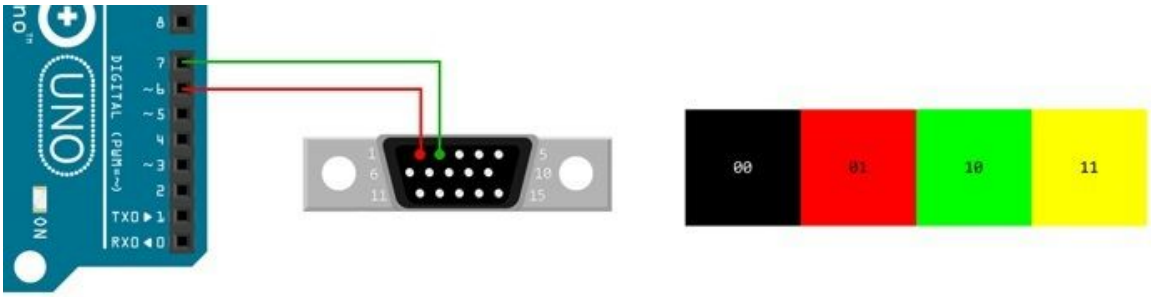
Do wyświetlania obrazu wykorzystywany jest ekran LCD, z którym komunikacja realizowana jest za pomocą kabla VGA. Informacje dotyczące koloru pikseli na ekranie przechowywane są w dwuwymiarowej tablicy kolorów o rozmiarze 120 na 60 jednostek. Każdy element tej tablicy przechowywany jest na dwóch bitach (co pozwala na uzyskanie czterech różnych kolorów na ekranie). Łączne wykorzystanie pamięci podręcznej tej tablicy wynosi ponad 1.75 KB na 2 KB dostępnej w Arduino Uno. Przy wyświetlaniu, tablica pikseli jest skalowana w górę, i wyświetlana w rozdzielczości 640x360. Każde pole w tablicy kolorów jest więc reprezentowane na ekranie przez prostokąt o rozmiarach około 5x6px.

Za przesyłanie obrazu i synchornizację pionową odpowiada oparta na Assemblerze biblioteka VGAX, zmieniona zgodnie z naszymi potrzebami.

Do wyświetlania pojedynczego piksela używamy funkcji `putpixel(byte x, byte y, byte color)` gdzie x i y oznaczają pozycję we wcześniej wspomnianych jednostkach. Color przyjmuje wartości od 0 do 3 oznaczające kolejno brak koloru(czarny), kolor1, kolor2 i wymieszanie.

Wyświetlany na ekranie kolor zależy od podłączenia pinów arduino 6 i 7 z odpowiednimi wejściami w złączu VGA w monitorze. W naszym programie używamy głównie funkcji `fillrect(byte x, byte y, byte width, byte height, byte color)` pozwalającej w łatwy sposób na narysowanie prostokąta zaczynającego się w punkcie (x,y) o długości boków width i height i kolorze color.

Zarówno putpixel jak i fillrect modyfikują tablicę kolorów przetwarzaną później i przesyłaną na ekran.



Wyjaśnienia działania

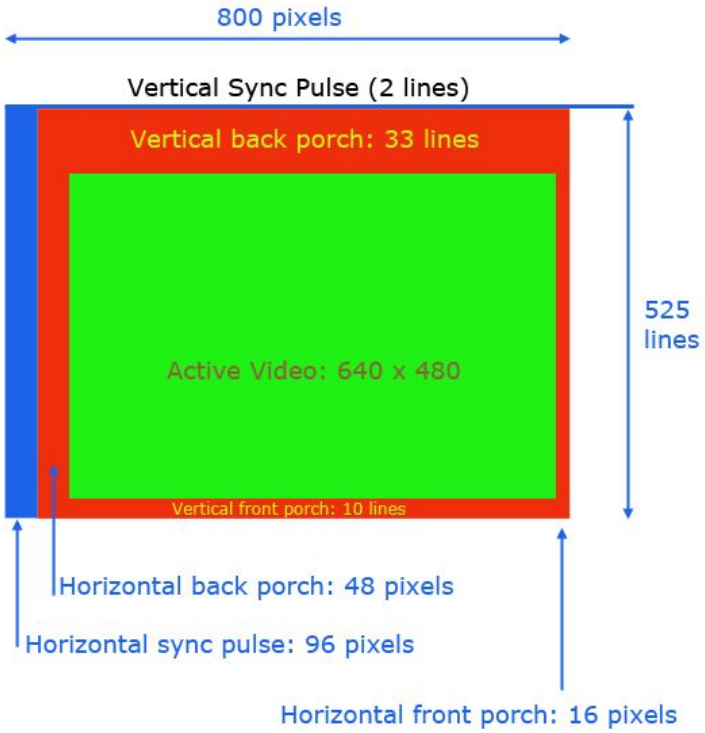
Aby zrozumieć jak to działa musimy zrozumieć jak działa taktowanie sygnału VGA.

W naszym projekcie istnieją 4 sygnały docierające go monitora.

- Synchronizacja pionowa - pin 9
- Synchronizacja pozioma - pin 3
- Czerwony analogowy - pin 6
- Zielony analogowy - pin 7

Trochę o starych monitorach i “gankach”.

Analizując, iż wyświetlamy obraz o rozdzielczości 640x480 starszy sprzęt(monitory CRT) potrzebuje chwili czasu aby przenieść elektrony. Stworzone zostały tak zwane ganki (porches).



Obraz nie pokrywa się wszystkimi wartościami z naszym programem.

Odświeżania ekranu zaczyna się z impulsu synchronizacji pionowej, który opowiada za zresetowanie góry ekranu. Ma na "back porch" liczbę linii przygotowanych do wyrysowania pustego obszaru na górze. Następnie rysuje obraz, i ma "front porch" dodatkowe linie do zwrócenia pustych informacji na dole ekranu.
Tymczasem, dla każdej linii istnieje impuls synchronizacji poziomej, który sygnalizuje początek tej linii, następnie inne opóźnienie, aby dać czas wiązce na przygotowanie się, a następnie rysuje linię i kilka dodatkowych ("front porch").

Synchronizacja pionowa.

- Częstotliwość odświeżania ekranu (np. 60 Hz)
- Rozdzielczość ekranu (np. 640 x 480)

Częstotliwość odświeżania to tempo, w którym cały ekran jest przerysowywany.
W naszym przypadku jest to 60Hz czyli 60 razy na sekundę.

Rozdzielczość jest to liczba (ilość na szerokość x ilość na wysokość) pikseli.

Począwszy od szybkości odświeżania 60 Hz, będzie to wymagało od nas impulsu "pionowej synchronizacji" 60 razy na sekundę, czyli co 1/60s (16.66 ms).

Generowanie sygnału synchronizacji pionowej.

Odbywa się ono za pomocą następującego fragmentu

```
//TIMER1 - vertical sync pulses
 pinMode(VSYNCPIN, OUTPUT);
 TCCR1A=bit(WGM10) | bit(WGM11) | bit(COM1B1);
  TCCR1B=bit(WGM12) | bit(WGM13) | bit(CS12) | bit(CS10); //1024
prescaler
 OCR1A=259; //16666 / 64 uS=260 (less one)
 OCR1B=0; //64 / 64 uS=1 (less one)
```

Wyjaśnienie linijek
Jak widać ustawiamy pin synchronizacji na wyjściowy, generator sygnału na Fast PWM, prescaler (spowolnienie taktowania) na 1024 przez co timer nalicza co 64uS.
Mając okres czasu 1/60s (16666 uS) musimy naliczać 260 razy(16666/64=260).
Jako, że w naliczaniu uzględniane jest 0 OCR1A ustawiamy na 259, następnie jako, że chce uzyskać czas pulsu modulacji bliską 64uS ustawiamy OCR1B na 0 co daje nam szerokość 2 linii(jedna linia to 1 / 60 / 525, co daje 31.7 uS) Ř

Synchronizacja pozioma.

Synchronizacja pozioma przekazuje monitorowi kiedy ma być rysowana każda linia.
Chcą ją obliczyć musimy podzielić częstotliwość odświeżania klatki przez całkowitą ilość linii co daje nam 31.74uS. Musimy także znać szerokość pulsu wiemy, iż na jeden cykl powstają 96 piksele. Chcemy znać szerokość na jeden piksel.
((1/60) / 525) / 800 = 39.68 nS // 800 jest to szerokość ze wszystkimi “gankami”

Tak więc szerokość całkowita pulsu synchronizacji poziomej to 96 * 39.68 nS = 3.8 uS

Generowanie impulsu synchronizacji pionowej

```
 pinMode(HSYNCPIN, OUTPUT);
 TCCR2A=bit(WGM20) | bit(WGM21) | bit(COM2B1); //pin3=COM2B1
 TCCR2B=bit(WGM22) | bit(CS21); //8 prescaler
 OCR2A=63; //32 / 0.5 uS=64 (less one)
 OCR2B=7; //4 / 0.5 uS=8 (less one)
```

Wyjaśnienie linijek:
Jak widać ustawiamy pin synchronizacji na wyjściowy, generator sygnału na Fast PWM, prescaler na 8 przez co timer nalicza co 0.5uS.
Potrzebujemy naliczać co 32uS (31.74 zaokrąglenie), OCR2A=63 (64 (32 / 0.5 = 64), liczymy 0), OCR2B = 7 ponieważ 3.8 uS w przybliżeniu 4 (4 / 0.5 = 8).

Generowanie pikseli

Do przechowywania informacji o pikselach używamy tablicy
vgaxfb[VGAX_HEIGHT*VGAX_BWIDTH] - wysokość, szerokość w “jednostkach”

Za wyświetlanie odpowiada:

ASSEMBLER

Wstawka assemblerowa stworzona za pomocą zapisu `asm volatile` zakończonego wpisem

```
: [port] "I" (_SFR_IO_ADDR(PORTD)),
"z" "I" (/*rline*/(byte*)vgaxfb + rlinecnt*VGAX_BWIDTH)
: "r16", "r17", "r20", "r21", "memory");
```

przez który są podane wartości do jej wnętrza.

```
"       ldi r20, 4      \n\t" //const for <<2bit
#ifdef VGAX_DEV_DEPRECATED
".rept 14       \n\t" //center line
"       nop             \n\t" //
".endr          \n\t" //
#endif
".rept 30       \n\t" //output 4 pixels for each iteration
"       ld r16, Z+      \n\t" //
"       out %[port], r16 \n\t" //write pixel 1
"       mul r16, r20    \n\t" //<<2 przesunięcie na następny element
"       out %[port], r0  \n\t" //write pixel 2
"       mul r0, r20     \n\t" //<<4 przesunięcie na następny element
```

```
"       out %[port], r0  \n\t" //write pixel 3
"       mul r0, r20      \n\t" //<<6 przesunięcie na następny element
"       out %[port], r0  \n\t" //write pixel 4
".endr          \n\t" //
"       nop              \n\t" //expand last pixel
"       ldi r16, 0       \n\t" //
"       out %[port], r16 \n\t" //write black for next pixels
```

Wyjaśnienie

Sztuczka polega na rozpakowaniu 4 pikseli z jednego bajtu danych poprzez wykonanie operacji bitowych przesunięć przed wysłaniem ich pojedynczo na PORTD.

Piksele pakowane są w formacie 0b11223344, pierwsza operacja zapisu piksela jest wstępnie przypisana do dwóch ostatnich bitów PORTD, gdzie połączone są “kable kolorów” VGA DSUB(pin 6 i 7, ponieważ PORTD przechowuje informacje o wszystkich pinach cyfrowych od 0 do 7).

Zamiast używać pętli używamy assemblerowego .rept który tworzy iterację kodu.

Potencjometr

Do wyznaczenia pozycji statku wykorzystywane jest wejście analogowe, do którego został podłączony potencjometr (do pinu **A2**). Sygnał analogowy przyjmuje wartości od 0 do 5V i jest przetwarzany przez przetwornik analogowo cyfrowy (ADC) na wartość liczbową **od 0 do 1023**.

Przetworzenie sygnału analogowego na cyfrowy zajmuje 13 cykli zegara przetwornika. W naszym programie przetwornik działa według domyślnych ustawień Arduino. ADC jest inicjalizowany automatycznie przy pierwszym użyciu funkcji bibliotecznej `analogRead`, wydłużając pierwszy odczyt do 25 cykli.

Zakres pozycji jakie przyjmuje statek gracza leży w zakresie od 7 do 112. Wzór na jej określenie przyjmuje więc następującą postać:

```
position = (analogRead(A2) * 0.1031f)+7;
```

Przycisk

Do wystrzelenia pocisku służy wejście cyfrowe, do którego został podłączony przycisk (do pinu o indeksie **13**). Pin 13 jest ustawiany jako pin wejściowy przy inicjalizacji systemu.

```
pinMode(13, INPUT);
```

Do odczytu wartości przekazywanej do pinu, wykorzystywana jest następująca funkcja:

```
bool currentButtonValue(){
     return digitalRead(13);
}
```

`digitalRead` zwraca stan wysoki (`HIGH`, `true`), gdy przycisk jest wciśnięty, zaś stan niski (`LOW`, `false`), gdy nie jest.

Zawsze przechowywana jest poprzednia wartość odczytana z przycisku. Czynności związane z wciśnięciem przycisku (`OnButtonPressed()`) są wywoływane, gdy podczas przejścia pętli programu okazuje się, że poprzednia wartość była negatywna, a obecna jest pozytywna.

```
if (currentButtonValue() == true && previousButtonValue == false){
     previousButtonValue = true;
     OnButtonPressed();
} else
     previousButtonValue = currentButtonValue();
```

Wciśnięcie przycisku po zniszczeniu rakiety gracza wywołuje restart gry.

Pamieć EEPROM

Przechowywanie informacji o poszczególnych dźwiękach w sekwencji melodycznej zużywa ogromną ilość pamięci podręcznej i może spowodować jej niedobór dla ważnych zmiennych. By obejść ten problem, wykorzystujemy dostępną na mikrokontrolerze pamięć EEPROM. Używany przez nas ATmega328P posiada jej 1Kb.
Dzięki temu, że EEPROM jest pamięcią nieulotną, byliśmy w stanie stworzyć program pomocniczy, za pomocą którego wgraliśmy do pamięci informacje o melodiach wygrywanych w grze. Do realizacji komunikacji z tą pamięcią wykorzystywana była biblioteka “EEPROM.h” udostępniającą wygodny interfejs do wykonywania wszelkich potrzebnych operacji.

EEPROM nie powinien być wykorzystywany do krótkiego przechowywania danych, bo po wykonaniu około 10000 operacji zapisu, pojawia się ryzyko uszkodzenia pamięci

Do wykonywania operacji na pamięci EEPROM wykorzystywane są między innymi rejestry:

- `EEAR` - address register. przechowuje informacje o adresie na którym wykonywane są aktualnie operacje
- `EEDR` - data register. przechowuje dane do zapisu/ odczytane
- `EECR` - control register. przechowuje informacje o wykonywanych akcjach. Na przykład bit 0 (`EERE`) wywołuje operacje odczytu wartości pod adresem `EEAR` do

`EEDR`, albo bit 1 (`EEPE`) wywołuje operacje zapisu wartości `EEDR` pod adresem `EEAR`.

Przerwania i zegary

W celu regularnej generacji sygnału wysyłanego do ekranu oraz do utrzymania stałej częstotliwości próbkowania dźwięku stosowane są przerwania oraz timery.

W celu uruchomienia przerwań korzystamy z timerów. Dla timerow 1, 2 i 3 istnieją odpowiednio rejestry: `TIMSK0`, `TIMSK1`, `TIMSK2`. Mamy na każdym z nich do dyspozycji po 3 bity dotyczące tego jaki rodzaj przerwań jest włączony. W naszym przypadku korzystaliśmy tylko z bitu zerowego (`TOIE0`, `TOIE1`, `TOIE2`), odpowiedzialnego za wywołanie przerwania po przepełnieniu odpowiedniego timera.
Rejestry dotyczące timerów dla wszystkich trzech wyglądają analogicznie, więc nie trzeba omawiać każdego z nich z osobna i wystarczy omówić najważniejsze rejestry jednego z nich:

- `TCCR0A` pozwala na ustalenie trybu pracy generatora sygnału taktującego. W naszym przypadku stosowany jest to “Fast PWM”
- `TCCR0B` posiada bity `CS00`, `CS01` i `CS02` wpływające na włączenie taktowania i jego ewentualne ustalenie prescalera, spowalniającego taktowanie. W naszym przypadku dla timera 1, prescaler wynosi 1024, a dla timera 2, wynosi 8.

Łączna szybkość taktowania timera da się przedstawić jako:

$F = F_0/N$

gdzie F_0 to szybkość taktowania mikrokontrolera, która w naszym przypadku wynosi 16MHz a N to wartość prescalera ustawiona w rejestrze `TCCR0B`

Timer0 i Timer2 mają rozmiary po 8 bitów każdy
Timer1 ma rozmiar 16 bitów

W celu uruchomienia lub wyłączenia przerwań, korzystamy odpowiednio z funkcji `sei()` oraz `cli()`, które zmieniają flagę I rejestru `SREG`

by przejąć zdefiniować czynności wykonywane podczas przerwania, stosujemy funkcje ISR() (od Interrupt Service Routine), w której podajemy instrukcje, które będą wykonywane w każdym przerwaniu. Jako argument funkcji podajemy nazwę przerwania. W naszym przypadku są to: `TIMER1_OVF_vect` i `TIMER2_OVF_vect`.

Głośnik

Biblioteka VGAX przejmuje kontrolę nad wszystkimi timerami, dlatego nie możemy stosować wbudowanej funkcji Arduino do wytwarzania dźwięku, opartej na zegarach. VGAX udostępnia jednak własną wersję tej funkcji, w następującej postaci:

```
void tone(unsigned int frequency) {
      afreq=1000000 / frequency / 2 / 32;
      afreq0=afreq;
}
```

Jak widać, zmienia on jedynie wartości dwóch zmiennych, do których odwołuje się później kod w Assemblerze. `afreq` przechowuje wygłaszany ton, zaś `afreq0` służy do szybkiego porównywania wartości z zerem we wstawce assemblerowej. Następnie odpowiednia wartość jest wysyłana do głośników za pośrednictwem assemblera (komenda `out`) przez pin **A0**, z pominięciem funkcji bibliotecznych, dzięki czemu proces jest przyspieszony, a synchronizacja obrazu nie zostaje zrujnowana.

W momencie inicjalizacji biblioteki VGAX, tryb działania pinu **A0** zostaje ustawiony na `OUTPUT`. Kod assemblerowy odpowiedzialny za modulację dźwięku, wywoływany jest podczas przerwania Timera2 (`ISR(TIMER2_OVF_vect)`).

ASSEMBLER

Wstawka assemblerowa jest stworzona za pomocą zapisu `asm volatile` zakończonego wpisem

```
: "z" (&afreq),
    [freq0] "r" (afreq0),
    [audiopin] "i" _SFR_IO_ADDR(PINC)
```

przez który są podane wartości do jej wnętrza.

1. Do rejestru R16 ładowana jest wartość zmiennej Z, przechowującej afreq. Następnie freq0 (przechowująca wartość afreq0) jest porównywana z zerem.

```
      ld r16, Z
      cpi %[freq0], 0
      breq no_audio
```

2. Jeżeli freq0 = 0 , następuje skok warunkowy (`breq`) do etykiety

```
      no_audio:
            nop
            nop
            nop
            nop
            nop
            nop
            rjmp end
```

gdzie dźwięk nie jest odtwarzany, ponieważ żądana częstotliwość jest zerowa.

3. W przeciwnym wypadku następuje załadowanie wartości częstotliwości do rejestru R18. Wartość R18 jest wysyłana do portu o numerze `audiopin`.

```
      out %[audiopin], r18
      rjmp end
```

Melodia jest na bieżąco ładowana z pamięci EEPROM, w metodzie `Audio.Update()` o następującej treści:

```
  void Update()
  {
      ++counter %= soundLength;
      if (!counter)
      {
        int ton = EEPROM[index+offset];
        ton*=toneHigh; //mnożenie wartości bajta z EEPROM-u do zakresu 0-1023
        VGAX::tone(ton);
        ++index %= leng;
      }
  }
```

gdzie `soundLength` oznacza długość pojedynczego dźwięku liczoną w przejściach głównej pętli programu. Po przekroczeniu tej wartości przez `counter`, z EEPROM-u ładowany jest nowy dźwięk. Klasa Audio cały czas przechowuje aktualny indeks tabeli dźwięków, który jest zerowany po przekroczeniu liczby dźwięków w zapisanym utworze. Domyślnie odtwarzany jest utwór *Für Elise* Beethovena. W pamięci zapisaliśmy 47 nut.

Kiedy następuje śmierć gracza, w głównej pętli wywoływana jest metoda `Audio.stroke()`, która zmienia melodię odtwarzaną przez komputer. `offset` jest wtedy zmieniony z 17 na 0, a `leng` (długość utworu) z 47 na 17 - odtwarzany jest inny utwór, zapisany w pamięci przed *Für Elise*.

Resetowanie

Z pinu cyfrowego **11** jest poprowadzony przewód do pinu o nazwie **RESET**. Port 11 inicjowany jest za pomocą dwóch instrukcji:

```
digitalWrite(11, HIGH);
pinMode(11, OUTPUT);
```

Zmieniamy tryb pracy pinu 11 na WYJŚCIE, jednak wcześniej musimy na niego wysłać sygnał wysoki. Gdyby w momencie zmiany trybu na wyjściu był sygnał niski, komputer zostałby zrestartowany natychmiast. Funkcja restartu, wywoływana po wciśnięciu Przycisku strzału po zniszczeniu rakiety gracza, wygląda następująco:

```
void reset(){
    digitalWrite(11, LOW);
}
```

Wystąpienie stanu niskiego powoduje wystąpienie obiegu prądu, wobec którego pin RESET natychmiast restartuje sprzęt.

Skutki awarii

Element	Skutki
Potencjometr	**ŚREDNIE**. Gracz traci kontrolę nad statkiem kosmicznym w grze. Zablokowana pozostaje więc jedna z podstawowych funkcjonalności gry.
Przycisk	**ŚREDNIE**. Gracz nie może strzelać, ale jest w stanie uciekać przed przeszkodami.
Potencjometr i przycisk jednocześnie	**KRYTYCZNE**. Gracz nie ma żadnego wpływu na rozgrywkę. Gra nie ma przez to sensu.
Ekran LCD	**KRYTYCZNE**. Gracz nie widzi, co się dzieje w grze.
Głośnik	**NIEGROŹNE**. Gracz jedynie nie słyszy muzyki, jednak muzyka jest tylko dodatkiem do właściwej rozgrywki.
Pamięć EEPROM	**NIEGROŹNE**. Istnieją różne warianty awarii: a) Uszkodzenie poprzez zbyt dużą liczbę zapisów: sprawi, że niemożliwy będzie ponowny zapis, jednak odczyt będzie wciąż możliwy i bezproblemowy, przynajmniej na niektórych bitach. Możliwe jest pojawienie się nieoczekiwanych bitów. b) Utrata funkcjonalności odczytu. Wtedy wszystkie “odczytywane” liczby będą zerami, przez co muzyka nie będzie odtwarzana.
Zegary	**KRYTYCZNE**. Uszkodzenie dowolnego timera uniemożliwia poprawne wyświetlanie obrazu oraz odtwarzanie muzyki. Z tego powodu sprzęt staje się bezużyteczny.

Program nie posiada mechanizmów diagnostycznych, ani naprawczych.