Politechnika Warszawska
Wydział Elektroniki I Technik Informacyjnych
Instytut Telekomunikacji

PSIR

Laboratorium

Ćwiczenie 2 2024.11.15
Temat: Podstawy komunikacji sieciowej z wykorzystaniem platformy Arduino i jej API

1.Wprowadzenie

Ćwiczenie to realizowane jest z wykorzystaniem platformy sprzętowej jak i emulowanej.

2.Instalacja i użytkowanie systemu kontroli wersji GIT

Istnieje na rynku cała plejada narzędzi do kontroli wersji oprogramowania, dla potrzeb studentów przygotowano system GIT. Dla systemu Windows można pobrać klienta z: `https://git-scm.com/download/win`, obecnie wspieranym i zalecanym jest „`Git-2.33.1-64-bit.exe`". Proszę narzędzie Git zainstalować najlepiej w katalogu `C:\Programs\Git`, używając domyślnych ustawień podczas procesu instalacji.
Dla systemów zgodnych z Linux w większości przypadków narzędzie Git jest dostępne w jego wbudowanych repozytoriach pakietów.
Przed rozpoczęciem pracy z systemem GIT należy w dowolnym miejscu utworzyć miejsce, np.: wydając polecenie CMD (aplikacja dostępna przez tzw. „lupkę") i przechodząc do katalogu za pomocą polecenia:

```
cd C:\Users\zsutguest\
```

a następnie utworzyć katalog np.:

```
mkdir temp
```

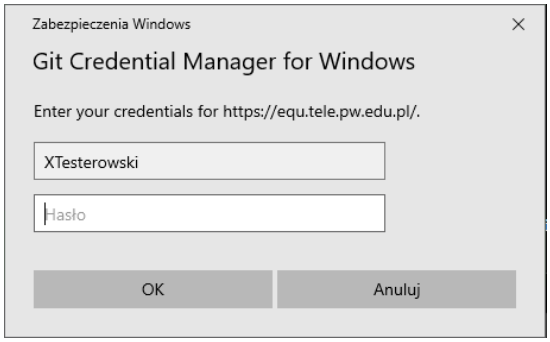
i wejść do niego:

```
cd temp
```

Mając gotowe miejsce możemy pobrać przygotowaną porcję plików klonując zdalne repozytorium (dane uwierzytelniające przesłane zostaną później, tu używany danych hipotetycznego i nie istniejącego użytkownika – tj. login: XTesterowski, haslo: Xtest2020):

```
git clone https://XTesterowski@equ.tele.pw.edu.pl/obir/XTesterowski
```

Podczas klonowania polecenie GIT zada pytanie o dane uwierzytelniające, które należy wpisać do okienka jak to:



Proszę pamiętać, że zachowanie systemu GIT może na niektórych instalacjach nieznacznie się różnić – np.: w systemie Linux pytanie o dane uwierzytelniające mogą być podawane w linii poleceń. Dodatkowo proszę pamiętać, aby nie używać metody kopiuj-wklej, z nieznanych przyczyn proces taki kończy się porażką autoryzacji.

Uwaga! polecenie „`git clone`” wykonuje się tylko raz w danej lokalizacji – czytaj zawsze w pustym miejscu, czyli tam gdzie nic jeszcze nie było wykonywane/modyfikowane.

Inne przydatne polecenia systemu GIT:
a)dodatnie wszystkich swoich poprawek do lokalnego repozytorium (domyślnie po sklonowaniu danych ze zdalnego repozytorium pracujemy na własnej kopii), korka w poleceniu oznacza – dodaj wszystkie zmodyfikowane pliki:

```
git add .
```

Po wydaniu polecenia możemy zobaczyć czy wszystkie nasze poprawione pliki zostały uwzględnione, wydając polecenie

```
git status
```

b)zatwierdzanie zbioru poprawek (niezbędne, aby system poprawnie rozpoznawał kiedy i co zostało uznane za zrobione):

```
git commit –a –m ”Zdawkowy i jednoznaczny opis co zostalo wykonane”
```

Po wydaniu powyższego polecenia możemy swoje poprawki i prace (nowe pliki) „wypchnąć” do zdalnego repozytorium:

```
git push
```

Więcej na temat systemu GIT można znaleźć w Internecie, w tym na stronie: https://git-scm.com

PLATFORMA SPRZETOWA (Arduino UNO)

A)Przygotowanie środowiska do pracy

Dostępne w laboratorium komputery m.in. Lenovo, działają pod systemem operacyjnym Windows 10 i posiadają zainstalowane pełne środowisko Arduino IDE (ikona na pulpicie).

Aby rozpocząć pracę z płytką podłączamy ją (otrzymaną od prowadzącego), za pomocą kabla „USB A – USB B”, do komputera. Poprawność podłączenia nowej platformy potwierdzamy, wyświetlając menu **`Tools->Port`**, w którym powinniśmy znaleźć informację o nowym porcie COM, z zaznaczeniem **`Arduino Uno`**.
Tak podłączoną platformę należy wybrać, a następnie potwierdzić poprawność je działania wybierając **`Tools->Get Board Info`**. W odpowiedzi, środowisko odczyta unikatowy numer seryjny podłączonej platformy. W razie problemów ze znalezieniem platformy, należy ją odłączyć i podłączyć ponownie, odczekując chwilę podczas której system operacyjny automatycznie wyszuka odpowiednie sterowniki.
Jeśli pomimo powyższych zabiegów, zaobserwujemy brak naszej platformy w spisie portów **`Tools->Port`**, może to oznaczać problem sprzętowy. Proszę zgłosić ten problem prowadzącemu.

B)Kompilacja i użytkowanie prostych programów przykładowych

Z każdym środowiskiem Arduino IDE dostarczany jest bogaty zestaw przykładów. Dostępne są one w **`File->Examples`**. Proszę wybrać przykład z **`01-Basic`** o nazwie **`Blink`**. Główny ekran środowiska przedstawia rys. 1.

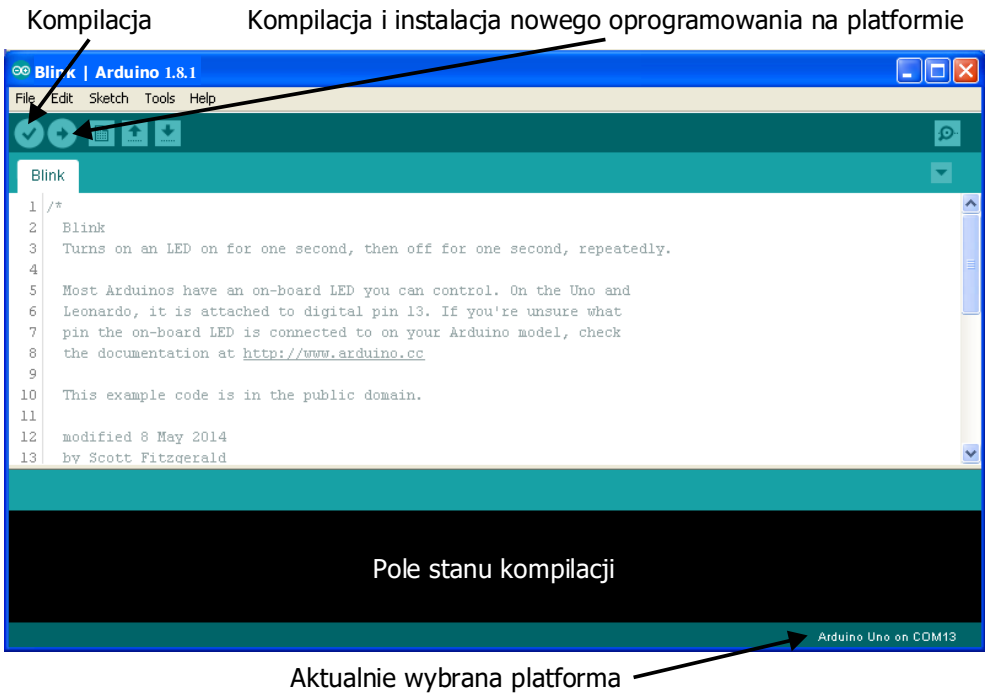
Po naciśnięciu przycisku ‘Kompilacja’ (rys.1), w polu stanu kompilacji prezentowany będzie przebieg tego procesu. Wszelkie błędy są także komunikowane w tym polu.

Aplikacja **`Blink`** (jak każda inna aplikacja w Arduino IDE) składa się ona z dwóch funkcji:

```
void setup(){...}
void loop(){...}
```

Pierwsza z tych funkcji jest wywoływana jednorazowo przez oprogramowanie systemowe zaraz po uruchomieniu i odpowiada za inicjalizację platformy Arduino. Druga jest cyklicznie wywoływana przez cały czas działania platformy Arduino po zakończeniu funkcji **`setup()`**. Czas przez jaki

procesor platformy wykonuje funkcję **`loop()`** powinien być jak najkrótszy, aby systemowe zadania 'tła' (np. obsługa niektórych interfejsów) mogły się właściwie wykonywać.
Po pomyślnej kompilacji (proces ukazuje pole stan kompilacji na rys.1) i załadowaniu kodu do procesora platformy Arduino UNO, działanie aplikacji rozpoczyna się automatycznie. Kod źródłowy programu **`Blink`** jest przedstawiony na listingu 1.

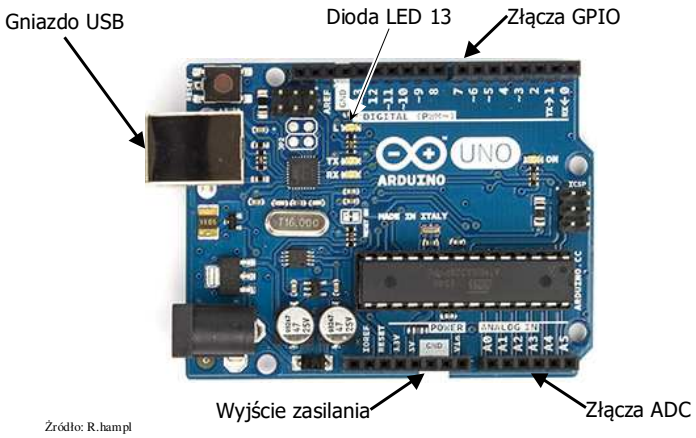


Rys. 1. Arduino IDE

Aplikacja **`Blink`** w funkcji **`setup()`** konfiguruje pin GPIO numer 13 jako wyjście. Natomiast funkcja **`loop()`** zmienia stan wyjścia GPIO numer 13; szybkość zmian wyznaczają wywołania funkcji **`delay()`**. Umiejscowienie diody LED podłączonej do wyjścia 13 pokazuje rys. 2.

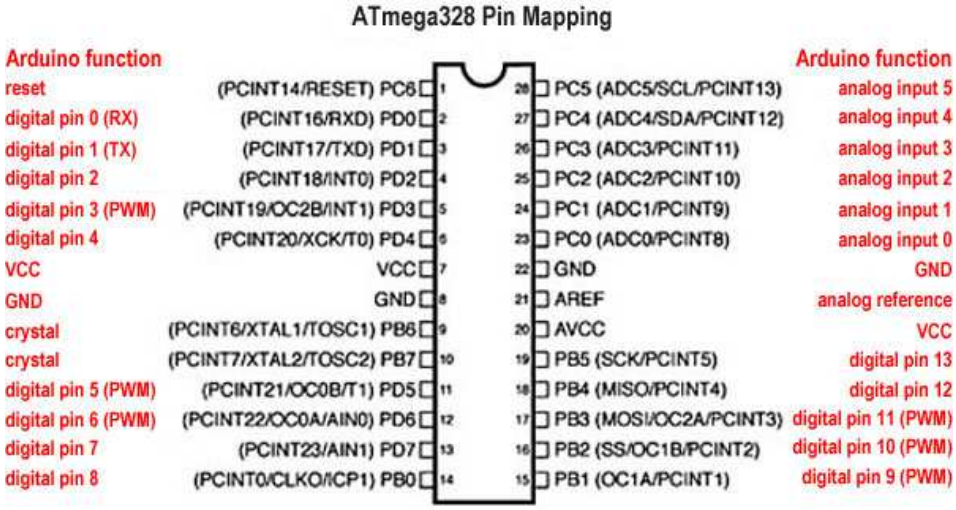
```
void setup() {
   pinMode(13, OUTPUT);      // initialize digital pin 13 as an output.
}
void loop() {
   digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
   delay(1000);              // wait for a second
   digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
   delay(1000);              // wait for a second
}
```

Listing 1. Aplikacja Blink



Rys. 2. Arduino UNO

Dla dalszej pracy z platformą Arduino UNO może być ważne odwzorowanie nazw końcówek mikrokontrolera Atmega328p w oznaczenia stosowane w bibliotekach zainstalowanych z Arduino IDE. Przyporządkowanie to przedstawia rys. 3.



Źródło: www.jameco.com

Rys.3. Mapowanie sygnałów Arduino UNO w Atmega328

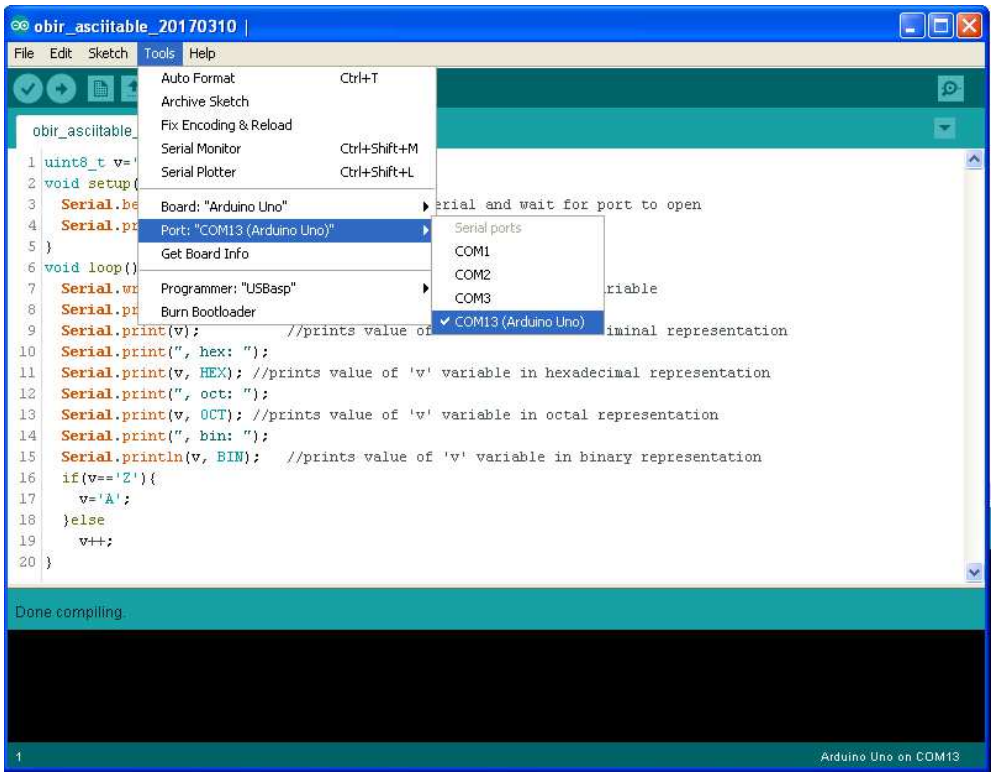
Drugim ważnym przykładem jak postępować z peryferiami platformy Arduino pokazuje program **`ASCIITable`**. Jest to próbka obsługi portu szeregowego. Na listingu 2 pokazano zmodyfikowaną wersję tego programu. Aplikacja ta stanowi test poprawności działania portu szeregowego. Dodatkowo, pokazuje on metody wypisywania danych na konsolę portu szeregowego.

```
uint8_t v='A';
void setup() {
   Serial.begin(115200);   //Initialize serial and wait for port to open
   Serial.println(F("ASCII Table ~ Character Map"));
}
void loop() {
   Serial.write(v);           //prints raw binary version of 'v' variable
   Serial.print(F(", dec: "));
   Serial.print(v);           //prints value of 'v' variable in decimal representation
   Serial.print(F(", hex: "));
   Serial.print(v, HEX);  //prints value of 'v' variable in hexadecimal representation
   Serial.print(F(", oct: "));
   Serial.print(v, OCT);      //prints value of 'v' variable in octal representation
   Serial.print(F(", bin: "));
   Serial.println(v, BIN);    //prints value of 'v' variable in binary representation
   if(v=='Z')
     v='A';
   else
     v++;
}
```

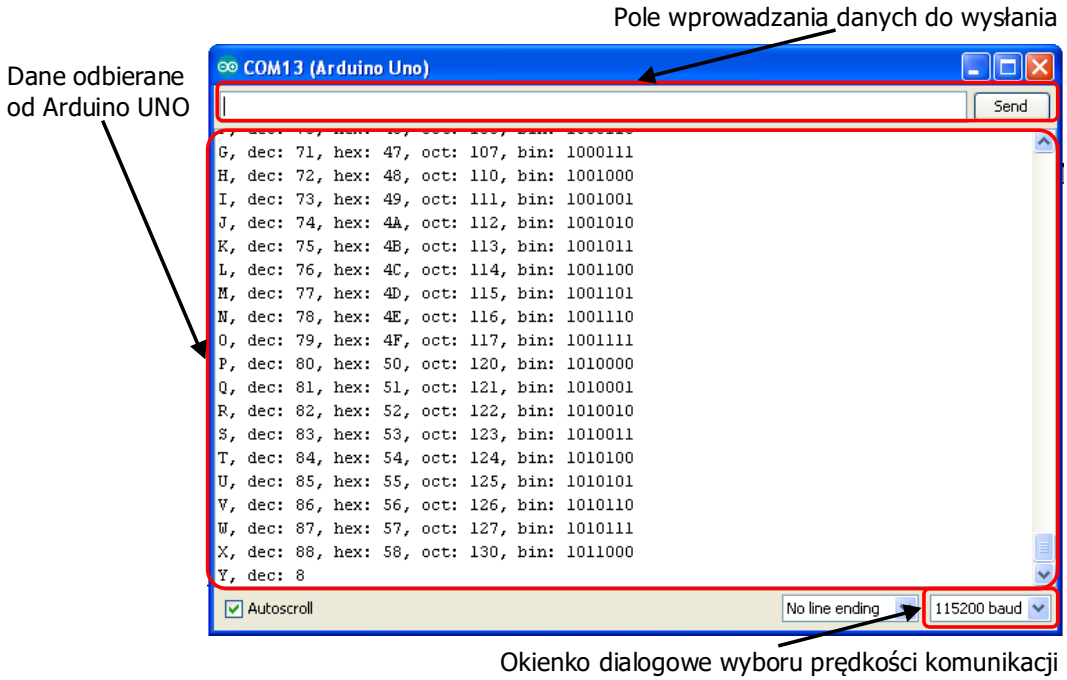
Listing 2. Program demonstrujący używanie portu szeregowego

Uruchomienie konsoli portu szeregowego w Arduino IDE składa się z (a) wybrania portu szeregowego, do którego podłączona jest platforma Arduino UNO (rys. 4), oraz (b) uruchomienia programu **`Serial Monitor`** za pomocą menu **`Tool`**. Uruchomioną konsolę portu szeregowego, połączoną z Arduino UNO, pokazuje rys. 5.

Nawiązując do listingu 2. trzeba nadmienić, że w szkicach można używać dwóch wersji metody **`print()`**. I tak **`Serial.print()`** to zwykłe wypisanie treści za pomocą portu szeregowego, a **`Serial.println()`** działa niemal identycznie z tym że wypisana treść kończona jest znakiem/znakami końca linii. Używanie odpowiedniej wersji pomaga lepiej kontrolować układ przekazywanych przez port szeregowy treści – sprawiać, że są lepiej czytelne.



Rys.4. Wybór portu komunikacji z platformą Arduino UNO



Rys. 5. Konsola portu szeregowego

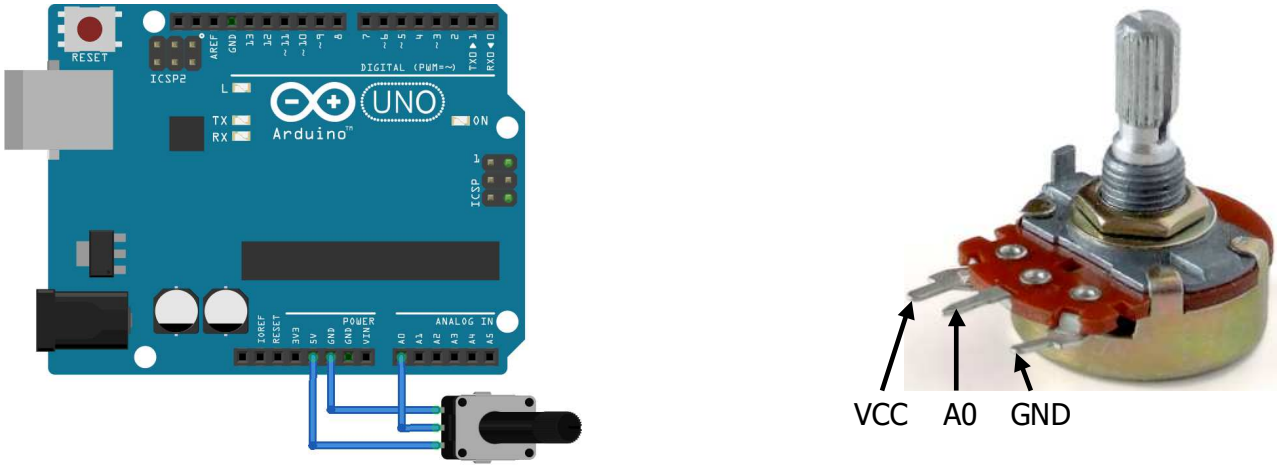
Wartym wyjaśnienia jest także zapis niektórych wywołań metody **`print()`**. Niektóre z nich zawierają nieco enigmatyczne wywołanie pomocniczej funkcji **`F(" ")`**. Jak widać w kodzie listingu 2. jest ona używana wyłącznie w fragmentach zawierających teksty o stałej treści przekazywanej do wypisania poprzez port szeregowy.

Zabieg ten w tak prostych przykładach mógłby być pominięty, jednakże w bardziej skomplikowanych aplikacjach pomaga rozwiązać problem zasobów pamięci RAM platformy Arduino UNO. Platforma ta jest zaopatrzona w tylko 2KB pamięci RAM. Architektura procesora zainstalowanego na tej platformie wymusza od kompilatora umieszczenie wszystkich danych (w tym także tekstów o stałej treści) w tej pamięci, nawet gdy niektóre z nich będą niezmienne przez

cały czas działania aplikacji. Używając funkcji **`F(" ")`** zmuszamy kompilator aby wybrane tak teksty umieszczał w pamięci kodu i używał do ich wypisania na porcie szeregowym metody **`print()/println()`** potrafiące z tej pamięci je pobrać.

C)Pomiar napięcia dołączonych peryferii

Arduino UNO posiada także możliwość pomiaru napięcia. Możliwe jest zatem mierzenie napięcia z wszelkiego rodzaju sensorów analogowych. Podczas zajęć laboratoryjnych mierzone jest napięcie z potencjometru obrotowego. Rysunek 6 pokazuje sposób podłączenia takiego potencjometru do używanej platformy.



Rys. 6. Dołączenie potencjometru do Arduino UNO oraz rozkład wyprowadzeń potencjometru

Aby zrealizować pomiar napięcia wystarczy wywołać (metodę) **`analogRead()`**, podając jako argument kod końcówki „analog input" (rys. 3). Listing 3 pokazuje jak to zrobić.

```
int sensorValue = 0;
void setup() {
   Serial.begin(115200);
}
void loop() {
   sensorValue = analogRead(A0);
   Serial.println(sensorValue);
   delay(2);
}
```

Listing 3. Pomiar napięcia za pomocą Arduino UNO

Wbudowany w mikrokontroler przetwornik ADC posiada 6 łatwych w użyciu wejść analogowych (A0, A1, A2, A3, A4, A5). Należy pamiętać, że funkcja **`analogRead()`** wykonuje pomiar blokując na czas konwersji kod aplikacji. Mimo, iż czas tej blokady nie jest zbyt długi należy o tym aspekcie pamiętać.
Rozdzielczość przetwornika to 10 bitów, a domyślnym napięciem referencyjnym jest 5V. Daje to poziom kwantyzacji równy około 4,9mV (5V/1024). Napięcie referencyjne można zmienić za pomocą funkcji **`analogReference()`**.

Podczas pracy z pomiarami napięć przydatna może być funkcja mapowania wartości z jednego zakresu w inny. Używamy do tego funkcji **`map()`**, której sposób użycia pokazuje listing 4.

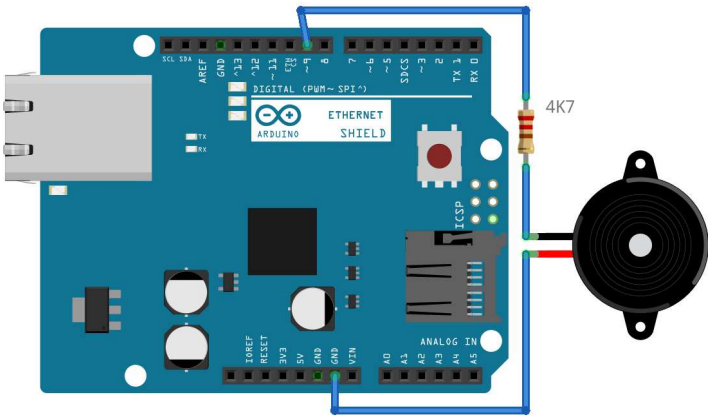
```
int v1=analogRead(A0);
uint8_t v2=map(v1, 0, 1023, 0, 255);
```

Listing 4. Użycie funkcji **`map()`**

Powyższy listing pokazuje jak w prosty sposób zamienić wartość odczytaną z przetwornika w wartość mieszczącą się na ośmiu bitach. Proszę pamiętać, że operacja ta sprawia że tracimy dokładność liczby zapisanej w zmiennej v2 względem oryginału zapisanego w v1.

D)Generowanie dźwięku

Podłączenie głośnika do Arduino ETH pokazano na rys. 7. Podczas laboratorium używa się głośnik piezzo-ceramicznego o nieco innej konstrukcji mechanicznej niż ta przedstawiona na tym rysunku. W razie wątpliwości proszę zapytać prowadzącego w celu zlokalizowania właściwego elementu.



Rys. 7. Podłączenie głośnika do Arduino UNO z nakładką ETH

Proste sterowanie głośnikiem pokazuje poniższy kod:

```
void setup() {
       pinMode(9, OUTPUT);
}
void loop() {
       digitalWrite(9, digitalRead(9) ^ 1);    // toggle state of pin 9
       delay(1);
}
```

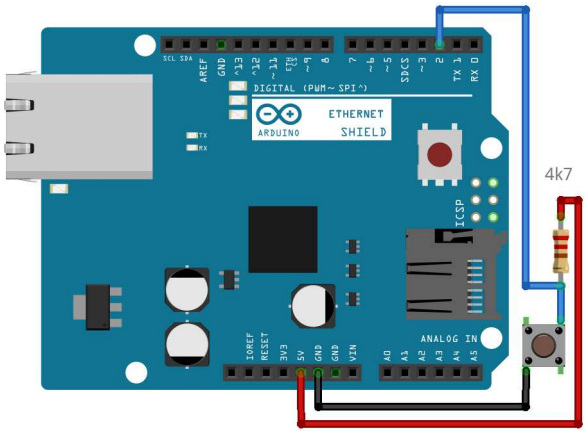
Należy zwrócić uwagę, że przedstawiony proces wymaga od procesora ciągłego sterowania membraną głośnika. Funkcja `digitalWrite()` jest odpowiedzialna za ruch membrany, zaś szybkość zmiany stanu pinu 9 wpływa bezpośrednio na ton generowanego dźwięku; w powyższym przykładzie będzie to około 500Hz. Sterowanie takie wymaga dość precyzyjnego odmierzania czasu. Zakłada się, że choć brak precyzji pomiaru czasu daje słyszalny efekt, to w pracach efekt ten można potraktować jako drugorzędny.

W powyższym przykładzie widać, że proces odmierzania czasu między akcjami związanymi z wprawianiem membrany głośnika w ruch jest nie precyzyjny i zajmuje czas procesora. Jednym ze sposobów radzenia sobie z tymi problemami jest zastosowanie liczników procesora ATMega328p wbudowanego w platformę Arduino UNO. Dzięki czemu jeden z tych liczników precyzyjnie odmierza czas po czym uruchomia procedurę obsługi przerwań w której nastąpi właściwa obsługa zmiany stanu membrany głośnika. Przykład poniższy zbudowano na podstawie informacji ze strony [http://playground.arduino.cc/Code/Timer1]:

```
#include "TimerOne.h"
void setup(){
       pinMode(9, OUTPUT);
       Timer1.initialize(1000);        // timer1 fires every 1000 microseconds
       Timer1.attachInterrupt(myisr); // when timer1 fires, myisr is executed
}
void myisr(){
       digitalWrite(9,digitalRead(9)^1); // actual tone generation
}
void loop(){
                                       // everything else is here
}
```

E)Obsługa włącznika chwilowego

Podłączenie włącznika chwilowego pokazuje rys. 8.



Rys. 8. Podłączenie włącznika chwilowego do Arduino UNO z nakładką ETH

W programie można sprawdzić stan takiego przełącznika dość prosto:

```
void setup() {
       pinMode(2, INPUT);
}
void loop() {
       if (digitalRead(2) == LOW){
              //switch is pressed
       }else{
              //switch is released
       }
}
```

Stosując podobne do powyższego podejście, należy pamiętać o problemie „drgania styków". W prostych aplikacjach zjawisko to może nie przeszkadzać, natomiast poprawna jego eliminacja jest dość trudna. Więcej na ten temat można znaleźć na stronie:

https://www.arduino.cc/en/tutorial/debounce

PLATFORMA EMULOWANA

A)Platforma emulowana

Dla potrzeb tego ćwiczenia laboratoryjnego została także przygotowana – podobnie jak dla ćwiczenia 1, maszyna wirtualna do pobrania z:

`https://secure.tele.pw.edu.pl/~apruszko/psir/psir23z_20230908_1052.ova`

Zawiera ona te same narzędzia jak te używane podczas ćwiczenia 1 oraz dodatkowo emulator **EBSimUnoEthCurses** oraz narzędzie PlatformIO. Proszę pamiętać, iż obraz pliku OVA dla tej maszyny wirtualnej zajmuje 1,3GB a na dysku twardym komputera gdzie zaimportowaną tę maszynę wirtualną zajmowana przestrzeń to 3,65GB. Maszyna wirtualna posiada dwa interfejsy sieciowe 'Adapter 1' podłączony jako interfejs NAT oraz 'Adapter 2' podłączony do wewnętrznej sieci „Host-only". Karta oznaczona jako 'Adapter 1' ma także skonfigurowane przekierowanie portu 2222 protokołu TCP maszyny goszczącej na port 22 protokołu 22 maszyny goszczonej. Dzięki temu przekierowaniu można połączyć się za pomocą protokołu SSH z poziomu maszyny goszczącej łącząc się z urządzeniem o IP: 127.0.0.1 na porcie 2222.

Uwaga! System operacyjny Windows w pewnych przypadkach blokuje poprzez swój wbudowany Firewall możliwość nawiązania połączenia z maszynami wirtualnymi działającym w tym trybie. Rozwiązaniem tego problemu jest wyłączenie tego Firewall'a (mnie bezpieczne) lub modyfikacja reguł przekazywania portów utrzymywanych przez VirtualBox.

Proszę pamiętać także, że w przypadku współpracy sprzętowego Arduino UNO z emulowanym komputerem PC (emulującym Arduino czy wykonującym aplikację Posix), oba elementy muszą działać w tej samej sieci IP. Przy ustawieniach wskazanych dla laboratorium 1. będzie to sieć podłączona do 'Adapter 1' którą należy przełączyć ją na tzw. interfejs mostkowany lub pozostawić

w trybie NAT ale wtedy konieczne jest ustawienie przekierowań dla używanych portów UDP (ich numer zależy od treści zadania otrzymanego do realizacji).
Prace z zagadnieniami sieciowymi z wykorzystaniem platformy Arduino UNO z nakładką Ethernet mogą być realizowane z wykorzystaniem symulatora **EBSimUnoEthCurses**. Jest to program, który emuluje CPU zainstalowane w Arduino UNO czyli Atmega328P na poziomie poszczególnych instrukcji tego procesora. Dla potrzeb dalszej części teksu można zdefiniować pojęcie „platforma emulowana" (czyli Arduino UNO z nakładką Ethernet) i „platforma emulująca" (komputer na którym będzie uruchamiany **EBSimUnoEthCurses**).
Proszę pamiętać, iż nie jest to idealna emulacja, istnieje wiele ograniczeń, wśród nich najważniejsze (poznane podczas testów i wywnioskowane z budowy):
-szybkość wykonywania instrukcji CPU nie odpowiada żadnej istniejącej realizacji krzemowej tego procesora – innymi słowy instrukcje te wykonywane są z szybkością będącą funkcją szybkości platformy na jakiej uruchomiono emulator (np.: Windows OS, Linux, ...),
-nie wszystkie zasoby i funkcje biblioteczne Arduino IDE oraz Platformio (opis wszystkich znajduje się pod: https://www.arduino.cc/reference/en/) są emulowane. Lista funkcji, które na platformie **EBSimUnoEthCurses** zachowują się nietypowo:
`delay()` – działanie nieprecyzyjne czasowo, zalecane jest stosowanie `ZsutMillis()`,
`millis()` – `delay()`, zatem zalecane jest stosowanie `ZsutMillis()`,
-niektórych funkcji nie zaleca się używać: `attachInterrupt()`, `detachInterrupt()`,
-niektóre zasoby peryferyjne wnętrza Atmega328P nie są emulowane: Timer 1 i 2 (Timer 0 domyślnie używane jest przez Arduino API do odmierzania czasu systemowego), oryginalny przetwornik ADC, generacja sygnałów PWM,
-dane wysyłane przez port szeregowy Ardunio UNO emulator wypisuje na konsoli w jakiej został uruchomiony, nie zwracając niemal uwagi na szybkość komunikacji portu szeregowego, dodatkowo warto pamiętać, że przetestowano w działaniu wyłącznie funkcje `Serial.begin()`, `Serial.print()` i `Serial.println()` – te ostatnie tylko z niektórymi typami danych używanych jako argumenty wywołania.
Platforma emulacyjna **EBSimUnoEthCurses** wspiera emulacje karty Ethernet w dość specyficzny sposób:
-emulacja sprowadza się do wyłącznie do komunikacji za pomocą protokołu UDP z użyciem wyłącznie klasy `ZsutEthernetUdp`,
-emulacja zakłada podanie jako parametru wywołania emulatora numeru IP spod jakiego odbywać się będzie komunikacja tym protokołem – jest to przydatne, gdy platforma emulująca zawiera wiele kart sieciowych o różnych numerach IP,
-komunikacja zapewniana przez `ZsutEthernetUdp` jest uproszczona[1] do obsługi jednego „strumienia" danych, innymi słowy aplikacja powinna ograniczyć się wyłącznie do wymiany datagramów z jednym zdalnym adresem IP i tylko na jednym numerze portu lokalnego – szczegóły stają się klarowne po przestudiowaniu dwóch dołączonych przykładów: `ZsutUdpNtpClient`, `ZsutEthUdpServer` – pliki te są umieszczone na serwerze studia, dla środowiska Arduino IDE muszą one mieć nazwy z rozszerzeniem `ino` a dla środowiska Platformio Core muszą mieć rozszerzenie `cpp`),
-inne dostarczone biblioteki są wspierającymi (np.: `ZsutFeatures` dostarcza dodatkowe usługi takie jak: `ZsutMillis()`, `ZsutDigitalRead`(), `ZsutAnalog0Read()`, `ZsutAnalog1Read()`, `ZsutAnalog2Read()`, `ZsutAnalog3Read()`, `ZsutAnalog4Read()`, `ZsutAnalog5Read()`, `ZsutIPAddress` dostarcza obsługę numerów IP, gdy pozostałe: `ZsutDhcp` i `ZsutEthernet` dostarczono dla zgodności z całym modelem programistycznym).
Warto nadmienić, że emulator nie wspiera poprawnie funkcji `digitalWrite`(). W jej miejsce i funkcji z nią pokrewnych takich jak `pinMode()`, `digitalRead()` należy stosować wywołanie funkcji: `ZsutPinMode()`, `ZsutDigitalWrite()` oraz `ZsutDigitalRead`().

[1] W obecnej wersji symulatora metoda write() klasy `ZsutEthernetUdp` nie zwraca poprawnie wyniku wykonanej operacji (zwraca zawsze 0). Zaleca się zignorowanie tego problemu – trwają prace z naprawą tego mechanizmu.

I tak `ZsutPinMode()` - ustawia tryb pracy określonego pinu, np.:

```
#define LED       ZSUT_PIN_D2
...
ZsutPinMode(LED, OUTPUT);
```

Natomiast `ZsutDigitalWrite()` – ustawia stan określonego pinu, np.:

```
ZsutDigitalWrite(LED, HIGH);
```

Dla odczytywania stanów pinów skonfigurowanych jako wejście używać trzeba funkcję `ZsutDigitalRead`(). Każdy bit zwracanej wartości odpowiada jednemu z bitów wejść – co pokazuje tabela:

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Pin	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Dodatkowo zdefiniowano piny emulowanego systemu: `ZSUT_PIN_D0`, `ZSUT_PIN_D1`, ..., `ZSUT_PIN_D13`, za pomocą tych definicji łatwiej tworzyć oprogramowanie – patrz plik blink.cpp w materiałach dodatkowych umieszczonych na serwerze studia.
Podczas emulacji stan wejść w określonej chwili czasu określa plik `infile.txt`. Ma on swoistą składnie. Jest to plik tekstowy o specyficznej gramatyce. Każda linia określa jedną definicję lub zdarzenie do wygenerowania przez symulator. W każdej linii można zapisać komentarz – rozpoczyna go znak ”#” i wszystko co jest po nim jest ignorowane przez emulator – nawet gdyby znak ”#” był na początku takiej linii – linia ta zostanie potraktowana w całości jako komentarz i nie będzie faktycznie wpływać na emulację. Poszczególne pola zapisane w liniach tego pliku mogą być rozdzielone separatorem jakim jest znak ”,” oraz dla czytelności jednym znakiem spacji.
Linie zaczynające się od znaku ”+” emulator traktuje jako wprowadzenie definicji określonego wejścia. Składania takiej definicji to:

```
+ nazwa_symboliczna,typ_zasobu,nazwa_fizycznego_zasobu
```

Gdzie:
`nazwa_symboliczna` – dowolna ciągła nazwa (bez spacji) o długości maksymalnie nie większej niż 64 znaki,
`typ_zasobu` – ciągła nazwa (bez spacji) o długości maksymalnie nie większej niż 32 znaki, dozwolone nazwy to: `quantity`, `status`, `action`.
`nazwa_fizycznego_zasobu` – ciągła nazwa (bez spacji) o długości maksymalnie nie większej niż 4 znaki, to pole może zawierać wyłącznie ciągi: `Z0`, `Z1`, `Z2`, `Z3`, `Z4`, `Z5`, `D0`, `D1`, `D2`, `D3`, `D4`, `D5`, `D6`, `D7`, `D8`, `D9`, `D10`, `D11`, `D12`, `D13` - opisują one fizycznie symulowane zasoby emulatora.
Linii z definicjami nie może być więcej niż liczba fizycznie definiowanych zasobów i zasadniczo nie powinno się definiować wielokrotnie tego samego zasobu fizycznego. Linie zaczynające się od znaku ”+” mogą występować wyłącznie w początkowej sekcji pliku `infile.txt`. Nie mogą być przeplatane z liniami z następnej sekcji.
Przykładowa linia z definicja to:

```
+ qTemperature,quantity,Z5    # sygnal wejsciowy przyjmujacy wartosci: 0 to -10C 1023 to +20C
```

W następnej sekcji mogą znajdować się linie zaczynające się od znaku „:” emulator traktuje jako wprowadzenie nowego stanu określonego wejścia. Składania takiej definicji to:

```
: czas_emulacji,nazwa_symboliczna,nowa_wartosc
```

Gdzie:
`czas_emulacji` – cyfry tworzące dodatnią liczbę w notacji dziesiętnej, będącą cyklem procesora w którym nowy stan określonego zasobu ma on przyjąć,
`nazwa_symboliczna` – nazwa zasobu (bez spacji) zdefiniowanego w poprzedniej sekcji, określająca zasób którego stan trzeba zmienić,

`nowa_wartosc` – cyfry tworzące dodatnią liczbę w notacji dziesiętnej, jako wartość którą ma przyjąć ten zasób, dla fizycznych zasobów o nazwach D0..D13 będzie to wartość 0 lub 1, dla zasobów Z0…Z5 będzie to dowolna wartość z zakresu 0…1023.

Przykładowa linia z definicja zmieniająca stan zasobu '`qTemperature`' (czyli fizycznego wejścia Z5) w cyklu 1250000, na wartość 100 mogła by wyglądać następująco:

```
: 12500000, qTemperature, 100
```

Wszystkie linie w sekcji zmian stanu (czyli zaczynające się od znaku ":") muszą być posortowane względem kolumny `czas_emulacji`, dozwolone jest nadawanie dwóm lub więcej liniom tego samego czasu emulacji. Brak zastosowania reguły posortowania względem kolumny `czas_emulacji` linii może uszkodzić emulacje i stan emulatora w danym momencie może być niezgodny z intuicją czy poprawnym działaniem.
Wszystkie znaki używane w pliku `infile.txt` to znaki z zakresu: [a-z], [A-Z], [0-9], [Spacja], "przecinek", "+", ":". Innymi słowy znaki z „czystego" ASCII[1], zabronione są polskie znaki, znaki nie drukowalne takie jak białe spacje itp. Linie powinny być kończone znakiem 0x0A czyli końcem linii zgodnym z sposobem kodowania końca linii w systemie Linux. Długość linii nie powinna być większa niż 1023 znaki. Przykład krótkiego pliku `infile.txt`:

```
#Opis zasobow
+ qTemperature,quantity,Z5    # input 0=-10 1023=+20
+ sOpening,status,D13         # input 0=OPEN, 1=CLOSED
#test nr.1
:  7500000, sOpening, 0
:  7500000, qTemperature, 0
: 10000000, qTemperature, 10
: 15000000, qTemperature, 1000
: 20000000, qTemperature, 10
: 22500000, qTemperature, 0
```

Proszę zwrócić uwagę na wielkość jaka jest zapisana w kolumnie `czas_emulacji` (liczby po znaku „:"). Emulator stosuje wartość tam zapisaną do wyznaczenia kiedy ma zmiana stanu w takiej linii zostać wykonana. Emulator zlicza każdą instrukcję wykonywana przez emulowany procesor. Na podstawie tego wie kiedy ewentualne zmiany jego wejść mają być uwzględnione. Proszę zauważyć, że fizyczny procesor stosowany w Arduino UNO czyli Atmega328P działa z zegarem 16MHz co odpowiada średnio około 16 milionom instrukcji na 1sek. Emulator nie daje jednak gwarancji wykonywania emulowanego oprogramowania z dokładnością czasu rzeczywistego. Niektóre fragmenty emulowanego kodu mogą wykonywać się szybciej inne wolniej. Proszę zauważyć, że w powyższym przykładzie pierwsza linia definiująca zmianę stanu wejść zapisana z czasem `7500000` wykonała się na komputerze prowadzącego laboratorium z opóźnieniem 6 sek. Proszę zatem tę nieokreśloność mieć na uwadze tworząc plik `infile.txt`. Jedynie `czas_emulacji` równy 0 dalej pewność, że zasób zmieni się w przewidywanym czasie – tuż przed rozpoczęciem emulacji.
Emulator podczas swojej pracy tworzy także plik z informacjami o zmianach stanów fizycznych zasobów We/Wy i czasie powstania tych zmian. Plik ten ma zdefiniowaną nazwę `outfile.txt` i ma bardzo podobną strukturę do pliku `infile.txt`. Nie zawiera on jednak linii z definicjami zasobów oraz stosuje nazewnictwo zasobów oparte wyłącznie o `nazwa_fizycznego_zasobu`. Oto przykład wygenerowanego podczas pracy emulatora pliku `outfile.txt`:

```
#Opening file to store genrated events (2023-11-03 11:39:33)
: 7500000, D13, 0
: 7500001, Z5, 0
: 10000000, Z5, 10
: 15000000, Z5, 1000
: 20000001, Z5, 10
: 22500000, Z5, 0
: 25000001, Z5, 1023
#Closing file with generated events (253858454)
```

Pierwsza linia pomaga zarządzać treścią takiego pliku – dla np.: archiwizacji. Ostatnia linia pomaga zorientować się ile cykli emulowanego procesora zostało wykonanych podczas działania emulatora. Warto także zauważyć, że poszczególne linie nie muszą w 100% odpowiadać danym z pliku `infile.txt`. Wynika to z tego, że emulator wykonuje zmianę określonego zasobu fizycznego z dokładnością do pojedynczej instrukcji emulowanego procesora – niektóre trwają 2 cykle zegarowe, stąd widać tę nie dokładność.

B)Integracja środowiska programistycznego dla potrzeb emulacji

Przed rozpoczęcie prac związanych z tworzeniem kodu dla Arduino UNO emulowanego przez **EBSimUnoEthCurses**, należy zainstalować Arduino IDE ze strony: https://www.arduino.cc/en/Main/Software (np.: dla systemu Windows: https://downloads.arduino.cc/arduino-1.8.16-windows.exe) lub Platformio Core ze strony: https://docs.platformio.org/en/latest//core/installation.html.

Instalacji Arduino IDE nie wykonuje się na maszynie wirtualnej `psir23z_20230908_1052.ova` a na komputerze goszczącym tę maszynę wirtualną.

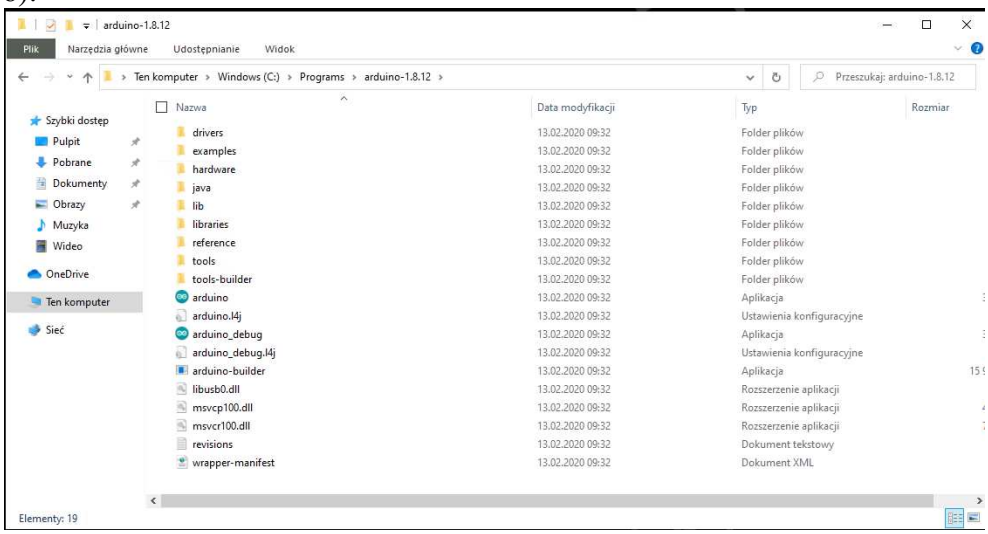
C)Instalacja środowiska Arduino IDE

Komputery laboratoryjne nie wymagają instalacji Arduino IDE, poniższa procedura opisuje przypadek gdy takowego środowiska brak.

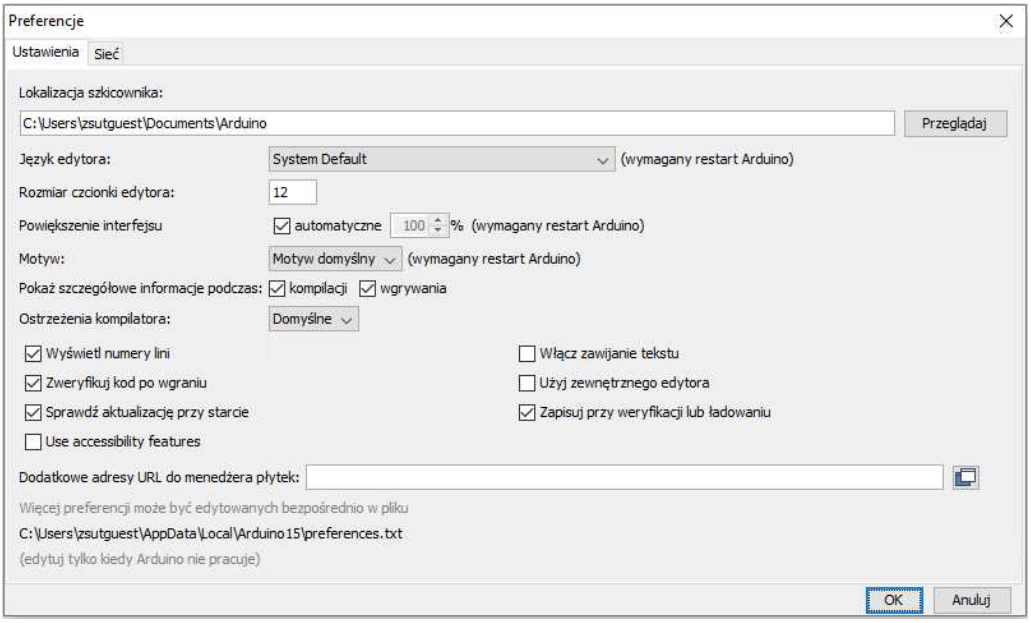
Zaleca się użycie instalatora z pliku EXE lub proste „rozpakowanie” pliku ZIP i gorąco odradzamy stosowanie wersji „Windows app” ze sklepu Microsoft – pojawiają się wtedy problemy z ustaleniem miejsca instalacji i przechowywania elementów składowych środowiska.

Zalecanym miejscem instalacji/kopiowania środowiska Arduino IDE jest `C:\Programs` (katalog, który należy wykreować samemu). Dziwić może, że nie jest to typowe miejsce czyli np.: „`C:\Program Files`”, ale wynika to głównie z dążenia do unikania spacji i znaków specjalnych w ścieżkach do wszelkich plików – takowe znaki bardzo często kompilują wiele operacji czy wręcz je uniemożliwiają.

Rysunek poniżej przedstawia widok plików po instalacji (dla Arduino 1.8.12 – co nie różni od wersji 1.8.16):



Po uruchomieniu Arduino IDE (plik: `C:\Programs\arduino-1.8.16\arduino.exe`) przed pierwszymi pracami należy skonfigurować i dodać biblioteki dla **EBSimUnoEthCurses**. Na początku zdobywamy wiedzę gdzie Arduino IDE będzie składował biblioteki, wybierając „File”->”Preferencje” zobaczymy obrazek:



Na rysunku widać, że w lokalizacji `C:\Users\zsutguest\Documents\Arduino`, będą przechowywane szkice, od pewnego wydania środowiska Arduino, również biblioteki są umieszczane w tym katalogu: `C:\Users\zsutguest\Documents\Arduino\libraries`. Do tego katalogu trzeba skopiować zawartość katalogu ZsutEthernet dostarczonego przez prowadzącego zajęcia laboratoryjne (serwer studia).

D)Instalacja środowiska Platformio Core

Komputery laboratoryjne nie wymagają instalacji Platoformio Code, domyślny użytkownik zsutguest tych komputerów ma gotowe do pracy środowisko. Poniższa procedura opisuje przypadek gdy takowego środowiska brak.

Aby móc korzystać z środowiska Platformio należy mieć zainstalowany Python Interpreter Python w wersji 3.6 lub nowszej. Detale jak zainstalować ten interpreter można znaleźć na stronie:

`https://docs.platformio.org/en/latest//faq.html#faq-install-python`

W większości przypadków instalacja Platformio Core sprowadza się do pobrania pliku:

`https://raw.githubusercontent.com/platformio/platformio-core-installer/master/get-platformio.py`

i wydania w linii poleceń jako zwykły użytkownik (instalacja nie wymaga żadnych specjalnych praw dostępu – ale działa tylko dla użytkownika który wykona tę instalacje), rozkazu:

```
python get-platformio.py
```

Instalacja wymaga pewnej przestrzeni dyskowej – należy mieć to na uwadze. Podczas instalacji wiele pakietów jest pobieranych z Internetu i mogą one zająć na dysku około 3GB.

E)Kopiowanie bibliotek wspierających tworzenie kodu dla platformy EBSimUnoEthCurses

Jak wcześniej wspomniano docelowe miejsce przechowania bibliotek w środowisku Ardunio IDE to: `C:\Users\zsutguest\Documents\Arduino\libraries`

Dla systemu Platformio Core powyższe biblioteki trzeba skopiować do katalogu `lib` w aktualnym katalogu roboczym (miejsce tworzenie oprogramowania w tym systemie). Dla celów dydaktycznych prowadzący zajęcia laboratoryjne udostępnia także dwa pliki w postaci źródłowej: `ZsutEthUdpServer` i `ZsutUdpNtpClient`. Stanowią one pomoc w testowaniu zestawionego środowiska, jak i w tworzeniu nowego kodu.

F)Kompilacja w środowisku Arduino IDE

Aby utworzyć plik z obrazem pamięci kodu (tzw. firmware) dla emulatora **EBSimUnoEthCurses**, trzeba jeden z tych plików wkleić do okienka głównego Arduino IDE a następnie wybrać „Narzędzia” -> „Płytka” -> „Arduino Uno” i dokonać kompilacji „Szkic” -> „Weryfikuj”.

Proces kompilacji umieszcza efekty swojej pracy w specyficznym miejscu, aby można było je znaleźć należy śledzić ten proces. Wybierając „Plik” -> „Preferencje” a następnie w polach koło

tekstu „Pokaż szczegółowe informacje podczas:” zaznaczając „kompilacji” i „wgrywania” (po czym wyłączamy i włączamy ponownie Arduino IDE), sprawimy, że Arduino IDE w dolnej części swojego okna będzie przedstawiać szczegóły kompilacji.
Arduino IDE przedstawia w dolnym oknie (na czarnym tle) proces kompilacji szkicu. W jednej z ostatnich linii system na ogół pokazuje tekst o treści podobnej do: „`Szkic używa 4132 bajtów (12%) pamięci programu. Maksimum to 32256 bajtów`”. Jest to bardzo cenna informacja o stopniu zużycia zasobów, jednakże na tym etapie bardziej interesująca jest dla nas jedna długa linia:

```
"C:\\Programs\\arduino-1.8.16\\hardware\\tools\\avr/bin/avr-size" -A
"C:\\Users\\ZSUTGU~1\\AppData\\Local\\Temp\\arduino_build_595773/ZsutUdpNtpClien
t.ino.elf"
```

W prawej części tej linii możemy przeczytać, że narzędzia kompilacji umieszczą pliki wynikowe w katalogu: `C:\\Users\\ZSUTGU~1\\AppData\\Local\\Temp\\arduino_build_595773`

Tam też trzeba szukać plików HEX niezbędnych dla dalszej pracy. Niestety notacja z dwoma znakami backslash („\\”) nie jest rozpoznawana poprawnie przez eksplorator systemu Windows, należy ją zamienić na tzw.: pojedyncze znaki backslash i tam szukać pliku, tj. w:
`C:\Users\ZSUTGU~1\AppData\Local\Temp\arduino_build_595773`

G)Kompilacja w środowisku Platformio

W przypadku systemu Platoformio jeden z przykładowych plików należy skopiować do katalogu `src` w aktywnym katalogu roboczym a następnie przemianować na main.cpp. Aby poddać kompilacji należy z linii poleceń wydać komendę:

```
platformio run
```

Po krótkiej chwili pojawią się komunikaty związane z procesem kompilacji. Podobnie jak w Arduino IDE efekty kompilacji umieszczany jest w dość specyficznym miejscu choć jest to zawsze to samo miejsce i jest nim: `.pio\build\uno` a właściwy wsad umieszczany jest w pliku `firmware.hex`. Dla przykładu gdyby nasz aktywny katalog roboczym był:
`C:\Users\apruszko\psir\app\ZsutUdpNtpClient\`
wtedy wynikowy plik miał by pełną nazwę i ścieżkę:
`C:\Users\apruszko\psir\app\ZsutUdpNtpClient\.pio\build\uno\firmware.hex`

H)Uruchomienie kompilatu

Po odnalezieniu pliku `ZsutUdpNtpClient.ino.hex` (lub `firmware.hex`), który zawiera obraz pamięci kodu, należy skopiować go do katalogu wewnątrz maszyny wirtualnej. Dzięki czemu program **EBSimUnoEthCurses** będzie miał do niego dostęp.

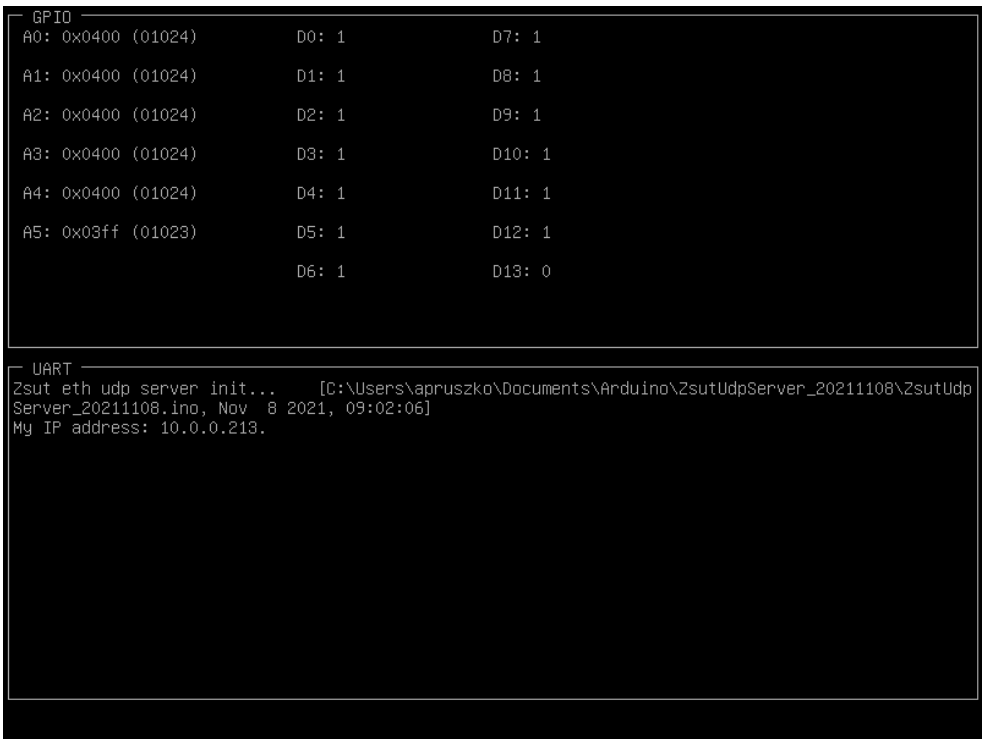
Właściwe wywołanie emulatora nastąpi poprzez:

```
EBSimUnoEthCurses -ip 192.168.89.3 ZsutUdpServer.hex
```

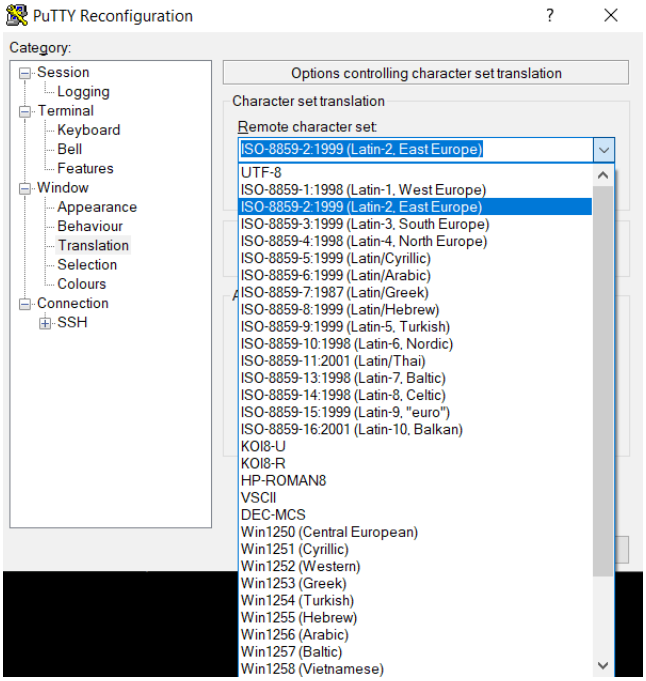
lub gdyby kopiowany był emulator z sieci (zakłada się że poniższe wywołanie którego skopiowano emulator):

```
./EBSimUnoEthCurses -ip 192.168.89.3 ZsutUdpServer.hex
```

Wywołanie powyższe należy zmodyfikować zgodnie z ustawieniami platformy emulującej, tj. numerem IP karty sieciowej maszyny wirtualnej (przypomnieć należy tutaj o konieczności zapewnienia łączności Arduino UNO oraz pozostałych elementów które mają współdziałać ze sobą). W powyższym przykładzie karta sieciowa ma `192.168.89.3`. Odkrycie numeru IP kary sieciowej można np.: wykonać z linii pleceń przez wydanie polecenia: „`ip a`”.
Po uruchomieniu emulatora zostanie otwarte okienko tekstowe co pokazuje poniższy obrazek (widok realnie działającego emulatora może się różnić, np.: nazwa pliku z rozszerzeniem „ino”):



Znany jest problem rysowania brzydkich ramek po powyższym wywołaniu narzędzia **EBSimUnoEthCurses**. Jest to związane z błędnym ustawieniem kodowania. W przypadku stosowania do połączenia się z maszyną wirtualną narzędzia Putty można zmienić w ustawieniach aktywnej sesji tzw. „Remote charakter set" z UTF-8 na ISO-8859-2:



Po uruchomieniu program **EBSimUnoEthCurses** emuluje on w nieskończoność program dla **Arduino UNO**, możliwe jest przerwanie jego pracy za pomocą klasycznego „CTRL-C".
Proszę pamiętać, że bardziej wyszukane używanie **EBSimUnoEthCurses** jest także możliwe, np.: uruchomienie dwóch lub więcej instancji tego programu. Jest to jednak możliwe, gdy każda z tych instancji będzie uruchamiana z plikiem HEX odpowiednio dla niej spreparowanym. W czasie preparowania takich plików HEX należy zwrócić uwagę na konieczność dzielenia się zasobami – np.: tutaj stosem IP. I tak gdy więcej niż jedna instancja emulatora będzie próbowała zestawić komunikację z wykorzystaniem protokołu UDP stosując ten sam numer lokalnego portu UDP, prawdopodobnie tylko pierwsza z nich otrzyma zasób sieciowy i wykona poprawnie emulowany

kod. Dla przykładu wsad: `ZsutUdpNtpClient.hex`, uruchomi się wyłącznie na pierwszej instancji emulatora, pozostałe instancje tego emulatora będą pracować błędnie, nie uda się im nic wysłać drogą sieciową.
Na dzień dzisiejszy emulator **EBSimUnoEthCurses** jest poprawnie działającym produktem klasy „pre-Beta". Tworząc własne szkice należy o tym pamiętać. Wszelkie uwagi odnośnie działania proszę kierować do prowadzącego zajęcia laboratoryjne, wstawiając na początku tematu słowo EBSimUnoEthCurses oraz w treści oprócz opisu problemu jako załączniki dołączając kod własnego szkicu oraz zrzut ekranu i danych wypisanych na konsoli (jeżeli takowe powstały). Po otrzymaniu takich danych będzie możliwa diagnoza ewentualnej usterki emulatora oraz jego poprawienie.

3.Współpraca z protokołem UDP/IP

a)Nakładka Ethernet dla Arduino UNO, zawiera układ `W5100`, który jest odpowiedzialny za komunikację poprzez łącze Ethernet.
Aby zapewnić komunikację poprzez protokół UDP za pomocą tej nakładki należy używać klasy `Ethernet` i `EthernetUdp`. W przypadku **EBSimUnoEthCurses** łączność ta jest emulowana, a odpowiednie funkcjonalności zapewniają klasy `ZsutEthernet` i `ZsutEthernetUdp`.
Dla zbudowania prostego serwera UDP, należy wykonać następujące kroki:
b)Dołączyć odpowiednie pliki nagłówkowe:

Arduino UNO	EBSimUnoEthCurses
`#include <Ethernet.h>` `#include <EthernetUdp.h>`	`#include <ZsutEthernet.h>` `#include <ZsutEthernetUdp.h>`

c)określić adres MAC (typowo programy raportujące ten numer podają go jako np.: 01:ff:aa:12:34:56) swojego urządzenia w 6 bajtowej tablicy `mac`, tutaj dla emulatora to:

```
byte mac[]={0x01, 0xff, 0xaa, 0x12, 0x34, 0x56};
```

Należy tu pamiętać, że używane numery MAC powinny być unikatowe – podczas pracy z emulatorem może wydawać się to nie potrzebne to jednak dla zgodności z oryginalną platformą należy o tym zawsze pamiętać gdyż w przeciwnym razie urządzenia podpięte do tej samej sieci mogą sobie nawzajem przeszkadzać podczas komunikacji.
d)utworzyć obiekt (globalny) typu `EthernetUDP/ZsutEthernetUDP` niezbędny do realizacji połączeń UDP:

Arduino UNO	EBSimUnoEthCurses
`EthernetUDP Udp;`	`ZsutEthernetUDP Udp;`

e)w funkcji `setup()` uruchomić bibliotekę `Ethernet/ZsutEthernet` oraz dla wygody podczas uruchamiania kodu można także przekazać na konsolę portu szeregowego otrzymany dla Arduino UNO z serwera DHCP adres IP:

Arduino UNO	EBSimUnoEthCurses
`Ethernet.begin(mac);` `Serial.println(Ethernet.localIP());`	`ZsutEthernet.begin(mac);` `Serial.println(ZsutEthernet.localIP());`

f)ustalić port UDP, inicjalizując globalną zmienną `localPort`. W przypadku uruchamiania równolegle wielu instancji **EBSimUnoEthCurses** to właśnie ten numer portu musi być każdej instancji ustalony indywidualnie.

```
unsigned int localPort = ...;
```

g)jednorazowo (np.: w ciele funkcji setup()), uruchomić obsługę protokołu UDP na porcie zapisanym w zmiennej `localPort`.

```
Udp.begin(localPort);
```

h)cyklicznie np.: w ciele funkcji **`loop()`**, przetwarzać pakiety UDP odebrane przez platformę kopiując ich treść do zmiennej `packetBuffer` o długości `MAX_BUFFER`:

```
unsigned char packetBuffer[MAX_BUFFER];
...
void loop(){
...
   int packetSize=Udp.parsePacket();
   if(packetSize){
      int r=Udp.read(packetBuffer, MAX_BUFFER);
      ...
   }
...
}
```

Metoda **`Udp.parsePacket()`** pomaga w obsłudze pakietów UDP i zwraca długość właśnie odebranego pakietu. Ważne odnotowania jest, iż to wywołanie nie czeka na dane; gdy ich nie ma, zwróci wartość 0, co będzie oznaczać, że nie odebrano żadnego pakietu. Dodatkowo należy pamiętać, iż wielkość datagramu zwrócona przez **`Udp.parsePacket()`** (tu skopiowana do zmiennej: `packetSize)` może być większa od wartości `MAX_BUFFER`, w takim przypadku programista musi „skonsumować" za pomocą metody **`Udp.read()`** także resztę datagramu lub mieć świadomość, iż traci część otrzymanych danych.
Gdy w logice aplikacji zajdzie potrzeba wysłać pakiet UDP z treścią przekazaną w zmiennej `sendBuffer` o długości `len` należy postępować następująco:

```
unsigned char sendBuffer[MAX_BUFFER];
int len=0;
...
sendBuffer[...]=...;
len=...;
...
Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
int r=Udp.write(sendBuffer, len);
Udp.endPacket();
```

Proszę pamiętać, że metoda **`Udp.write()`** powinna zwracać liczbę przyjętych przez warstwę niższą danych[2] do wysłania (tu: zmienna '`r`'), w niektórych przypadkach liczba ta może być mniejsza od zmiennej '`len`', wtedy programista także musi pamiętać o rozwiązaniu tego przypadku i ewentualnym przesłaniu pozostałej części danych.
Dodatkowo proszę pamiętać, że `Udp.remoteIP()`, `Udp.remotePort`() – mogą zwrócić wartości błędne w przypadku gdy węzeł nie otrzymał żadnego datagramu UDP lub gdy w trakcie działania programu węzeł otrzymał nowy pakiet. W takim przypadku należy w kodzie ustalić adres IP i numer portu zdalnego węzła do którego pragniemy wysłać datagram UDP a potem podjąć próbę wysłania danych, dla przykładu może to wyglądać tak:

Arduino UNO

```
#define UDP_REMOTE_PORT       4321
#define PACKET_BUFFER_SIZE    32
IPAddress address_ip=IPAddress(10,0,4,1);
unsigned char sendBuffer[PACKET_BUFFER_SIZE];
...
sendBuffer[...]=...;
int len=...;
...
Udp.beginPacket(address_ip, UDP_REMOTE_PORT);
int r=Udp.write(sendBuffer, len);
Udp.endPacket();
```

[2] W **EBSimUnoEthCurses** metoda write() klasy `ZsutEthernetUdp` nie jest poprawnie emulowana i nie zwraca poprawnie wyniku.

EBSimUnoEthCurses

```
#define UDP_REMOTE_PORT       4321
#define PACKET_BUFFER_SIZE    32
ZsutIPAddress address_ip=ZsutIPAddress(10,0,4,1);
unsigned char sendBuffer[PACKET_BUFFER_SIZE];
...
sendBuffer[...]=...;
int len=...;
...
Udp.beginPacket(address_ip, UDP_REMOTE_PORT);
int r=Udp.write(sendBuffer, len);
Udp.endPacket();
```

Po wywołaniu ostatniej funkcji (endPacket) datagram UDP zostanie wysłany do maszyny o IP: 10.0.4.1 na jej port UDP numer 4321.

8. Literatura

1.Kodowanie ASCII https://en.wikipedia.org/wiki/ASCII, ostatnia wizyta 2024.11.02