Dokument powstał w ramach pracy na projektem: Unowocześnienie treści wykładowych i ćwiczeń laboratoryjnych z oferty dydaktycznej Zakładu Sieci i Usług Teleinformatycznych (CPR-IDUB/298/Z15/2024)

PSIR

Laboratorium

Ćwiczenie 2 2025.11.03
Temat: Podstawy komunikacji sieciowej z wykorzystaniem platformy Arduino ESP32 i emulatora Arduino

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-2.50.0-64-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.:

```
md temp
```

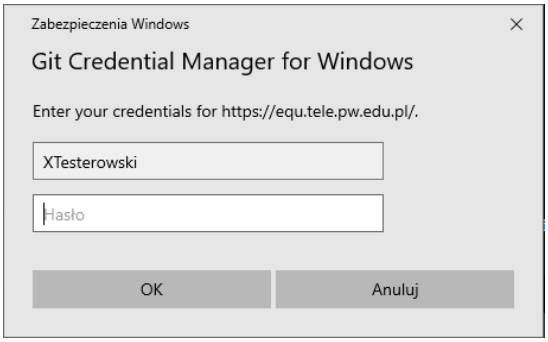
i wejść do niego:

```
cd temp
```

Mając gotowe miejsce możemy pobrać przygotowaną porcję plików klonując zdalne repozytorium (dane uwierzytelniające przesłane zostaną listem, tu używany danych przykładowych nie istniejącego użytkownika – tj. login: tester, hasło: tester_haslo):

```
git clone https://tester@equ.tele.pw.edu.pl/pasir/tester
```

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. Warto

tu zaznaczyć że używanie metody kopiuj-wklej może zakończyć się porażką, jest to związane z specyficznymi konwersjami dokonywanymi w niektórych systemach operacyjnych podczas takich operacji.
Uwaga! polecenie „`git clone`" wykonuje się tylko raz w danej lokalizacji – co oznacza że konieczne jest wykonanie tej operacji w pustym miejscu, czyli tam gdzie nic jeszcze nie było wykonywane lub modyfikowane.
Przydatne polecenia systemu GIT:
a)Przyjmijmy, że wytworzyliśmy plik main.c i chcemy dodać go do lokalnego repozytorium GIT, wydając polecenie:

```
git status
```

powinniśmy zobaczyć:

```
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        main.c

nothing added to commit but untracked files present (use "git add" to track)
```

Proszę zwrócić szczególną uwagę na kolor jaki został użyty do wypisania main.c – czerwony ma przyciągnąć naszą uwagę, wskazując tu że system GIT w tym katalogu nic nie wie o tym pliku (untracked). Co można zrobić – dodać taki plik do repozytorium:

```
git add main.c
```

Gdybyśmy chcieli dodać wiele plików czy całych podkatalogów w aktualnym miejscu, lub gdybyśmy wykonali poprawki plików w aktualnym katalogu wydajemy polecenie (kropka w poleceniu oznacza – dodaj wszystkie zmodyfikowane pliki w aktualnym katalogu:

```
git add .
```

Po wydaniu polecenia dodawania, możemy zobaczyć czy wszystkie nasze poprawione pliki zostały uwzględnione, wydając polecenie git status:

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   main.c
```

Teraz widać, że plik main.c jest dodany do repozytorium lokalnego. Aby jednak wprowadzone zmiany były zatwierdzone wykonujemy:

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

A narzędzie GIT zaraportuje:

```
[master (root-commit) 7344c97] Zdawkowy i jednoznaczny opis co zostalo wykonane
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 main.c
```

Na tym etapie wydając polecenie git status zobaczymy:

```
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)

nothing to commit, working tree clean
```

Na komputerach laboratoryjnych może zdarzyć się że po poleceniu `git commit` zobaczymy:

```
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'zsutguest@ZsutLap21.(none)')
```

Jest to normalne zachowanie na komputerach gdzie nie pracowano jeszcze z GIT.

```
git config user.email "tester@jego.poprawna.domena"
git config user.name "Tester"
```

Git zapamięta tego użytkownika i będzie go kojarzył z wszelkimi pracami. Po powyższym ustawieniu należy ponownie wydać polecenie git commit (opisane nieco wcześniej) a zobaczymy wydając polecenie git log:

```
commit 2587f3f4746b710552744e0a61555e4c7827e85f (HEAD -> master)
Author: Tester <tester@jego.poprawna.domena>
Date:   Fri Jun 27 09:36:51 2025 +0200

    Zdawkowy i jednoznaczny opis co zostalo wykonane
```

Dla prac wyłącznie lokalnych prace z GIT można by zakończyć – ale naszej pracy nie będzie mógł obejrzeć nikt poza nami, a wymagane jest aby prowadzący zajęcia w laboratorium mieli wgląd do wyników prac studentów. Aby zrealizować tę operację wydajemy polecenie wypchnięcia danych na zdalne repozytorium:

```
git push
```

Zobaczymy:

```
Fetching remote heads...
  refs/
  refs/heads/
  refs/tags/
updating 'refs/heads/master'
  from 0000000000000000000000000000000000000000
  to   2587f3f4746b710552744e0a61555e4c7827e85f
    sending 3 objects
    done
Updating remote server info
To https://equ.tele.pw.edu.pl/pasir25Z/tester
 * [new branch]      master -> master
```

Pytaniem może być – a co zobaczy prowadzący zajęcia gdy zajrzy do repozytorium:

```
commit 2587f3f4746b...55e4c7827e85f (HEAD -> master, origin/master, origin/HEAD)
Author: Tester <tester@jego.poprawna.domena>
Date:   Fri Jun 27 09:36:51 2025 +0200
Zdawkowy i jednoznaczny opis co zostalo wykonane
```

Kluczowe jest to, żeby obie strony widziały ten sam numer commit'u (tutaj: 2587f3f4746b710552744e0a61555e4c7827e85f) oraz co ważniejsze pracowały na wersji "master" (tu pokazano ten fakt jako: HEAD -> master, origin/master, origin/HEAD).

Proszę pamiętać, że istnieje metoda na sprawdzenie czy na zdalnym repozytorium git jest ta sama zawartość co na naszym lokalnym repozytorium – wystarczy sklonować zdalne repozytorium w

innym pustym miejscu i porównać zawartości, lub przynajmniej czy tzw. numery commit są identyczne.

Czasami podczas prac ze względu na oszczędność miejsca na dyskach, konieczne jest pominięcie plików wynikowych lub tajnych (klucze i dane uwierzytelniające). Przydanym do tego jest plik .gitignore w głównym katalogu lokalnego repozytorium lub w jednym z jego podkatalogów (jego działanie będzie odnosiło się tylko do podkatalogów z miejsca umieszczenia tego pliku). Kreujemy go poprzez CMD poleceniem (Windows):

```
type NUL > .gitignore
```

Linux:

```
touch .gitignore
```

Po wykreowaniu tego pliku – koniecznie trzeba dodać go do GIT:

```
git add .gitignore
```

Od tego momentu do wnętrza tego pliku wpisujemy nazwy katalogów i plików które chcemy aby GIT pomijał. Dla przykładu gdyby podczas prac powstawały w katalogu o nazwie np.: wynik pliki wynikowe o nazwie np.: test i chcielibyśmy aby GIT ignorował ten plik możemy wpisać do .gitignore wpis (dla Linux):

```
wynik\test
```

Dla Windows:

```
wynik/test
```

Więcej informacji o GIT można znaleźć na stronach:

https://git-scm.com/doc
https://git-scm.com/docs/git
https://git-scm.com/docs
https://www.kernel.org/pub/software/scm/git/docs/git.html

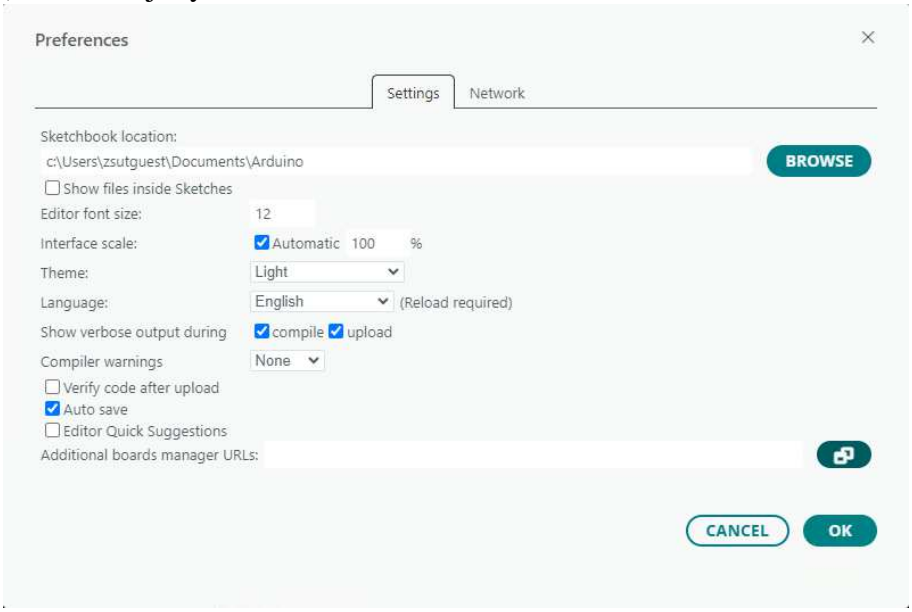
Wprowadzenie w aspekty związane ze środowiskiem programistycznym

A)Przygotowanie środowiska do pracy (ESP32, EBSimUnoEthCurses)

UWAGA! Ten etap wykonywany jest tylko raz – Student/Studentka przed jego wykonaniem powinni sprawdzić czy odpowiednie wsparcie nie zostało w środowisku zainstalowane.

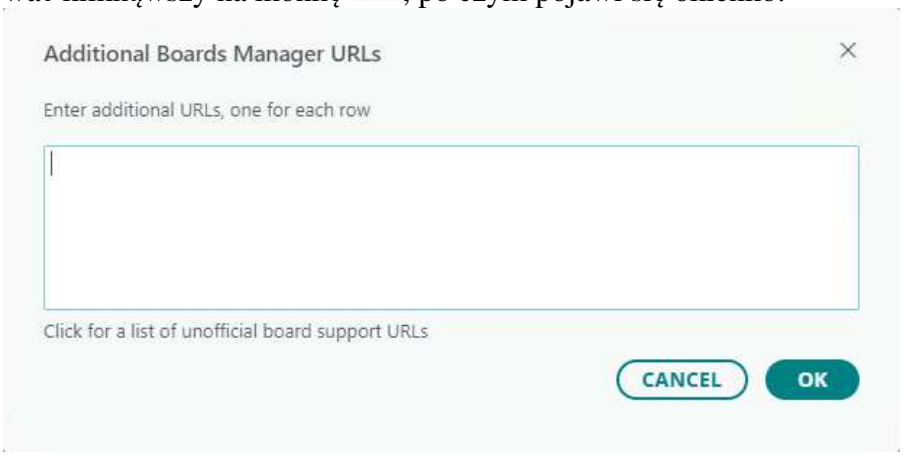
Poniższy tekst jest wprowadzeniem w używanie środowiska Arduino IDE. Platforma **EBSimUnoEthCurses** jest w dużym stopniu zgodna z Arduino UNO a ta jest domyślnie wspierana przez to środowisko – więc poniższy opis będzie ukierunkowany głównie na platformę zgodną z ESP32.

Dostępne w laboratorium komputery, działają pod systemem operacyjnym Windows 10 i posiadają zainstalowane pełne środowisko Arduino IDE (ikona na pulpicie). W ramach zajęć laboratoryjnych używana jest platforma ESP32 na której można instalować oprogramowanie binarne (ang. firmware) będące wynikiem kompilacji źródeł zgodnych z tzw. modelem programistycznym Arduino (ang. Arduino Framework). Aby Arduino IDE mogło tworzyć wspomniane pliki binarne konieczne jest dodanie obsługi płytek ESP32. Realizuje się to za pomocą wyboru okienka preferencji (**`Files -> Preferences`**), co ukazuje rysunek:



Oprócz wielu opcji widać, że ustawiono w opcjach „Show verbose output during" „compile" oraz „upload" – są to bardzo przydatne elementy, pomagają podczas prac programistycznych ukazując szczegóły pracy systemu ukazując jakie polecenia są uruchamiane oraz z jakimi parametrami.

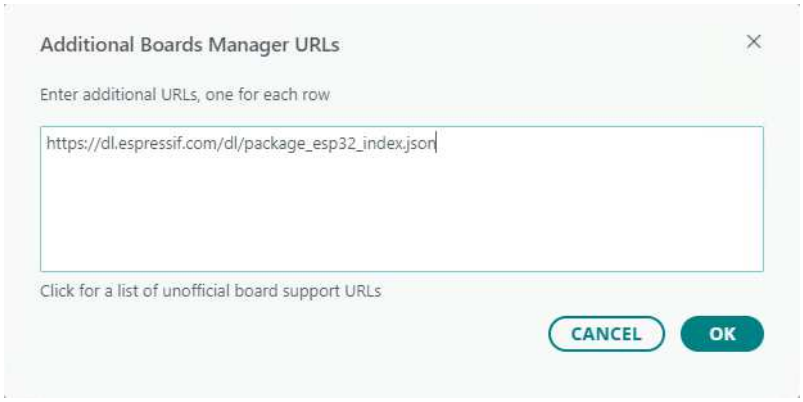
Dla dostosowania środowiska do pracy z urządzeniami ESP32 najważniejsze jest dodanie informacji o lokalizacji dodatkowych informacji dla menadżera płytek („Additional board manager URLs"), co można zrealizować kliknąwszy na ikonkę , po czym pojawi się okienko:



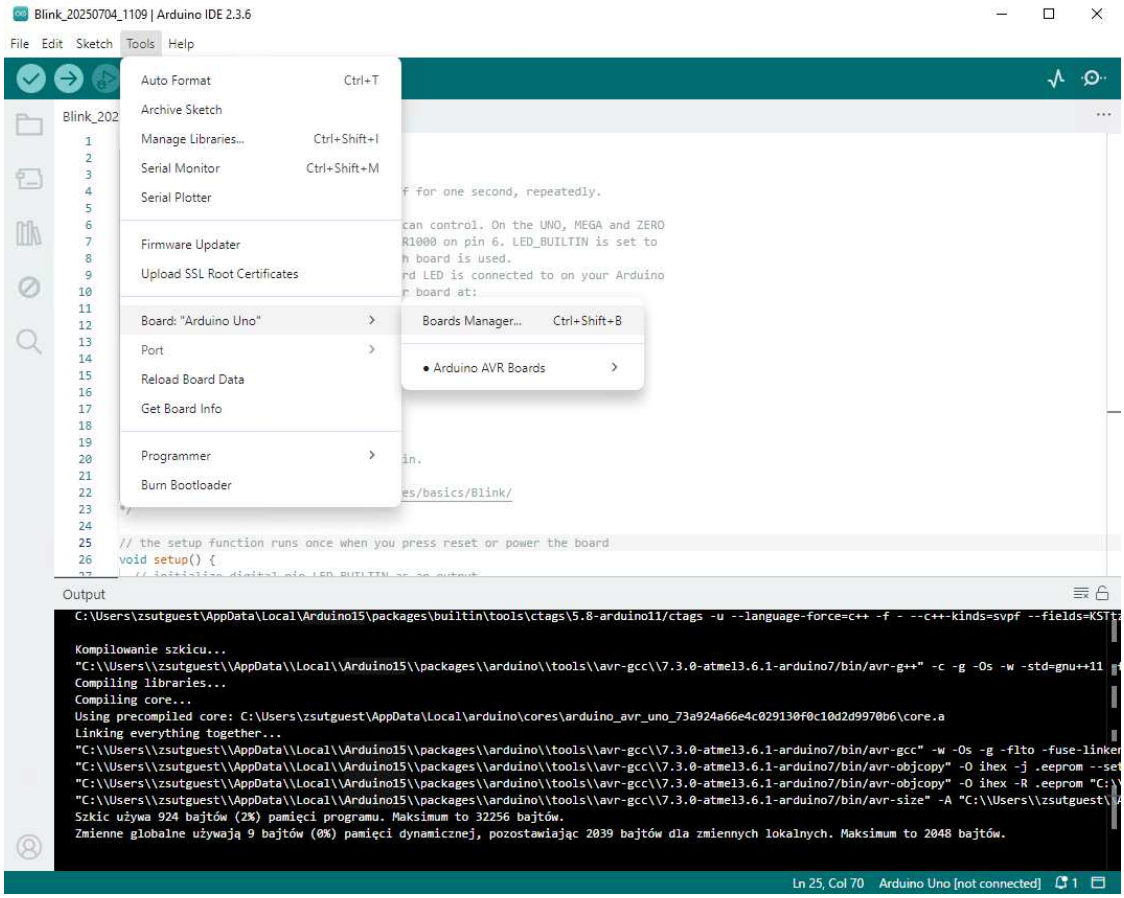
W polu opisanym jako „Enter additional URLs…" można wpisać lokalizacje wymaganych informacji. Stosowane płytki wymagają informacji ze strony:

`https://dl.espressif.com/dl/package_esp32_index.json`

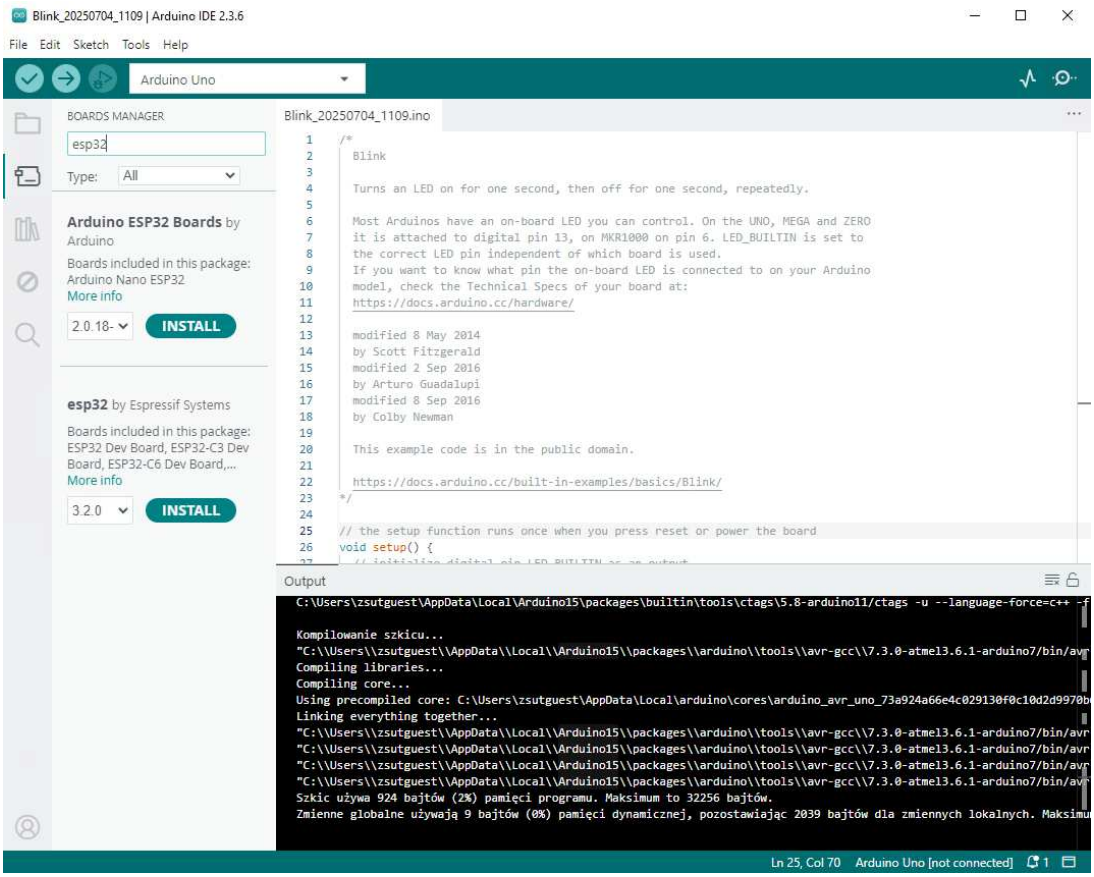
Rysunek poniżej pokazuje jak wpisano ten ciąg:



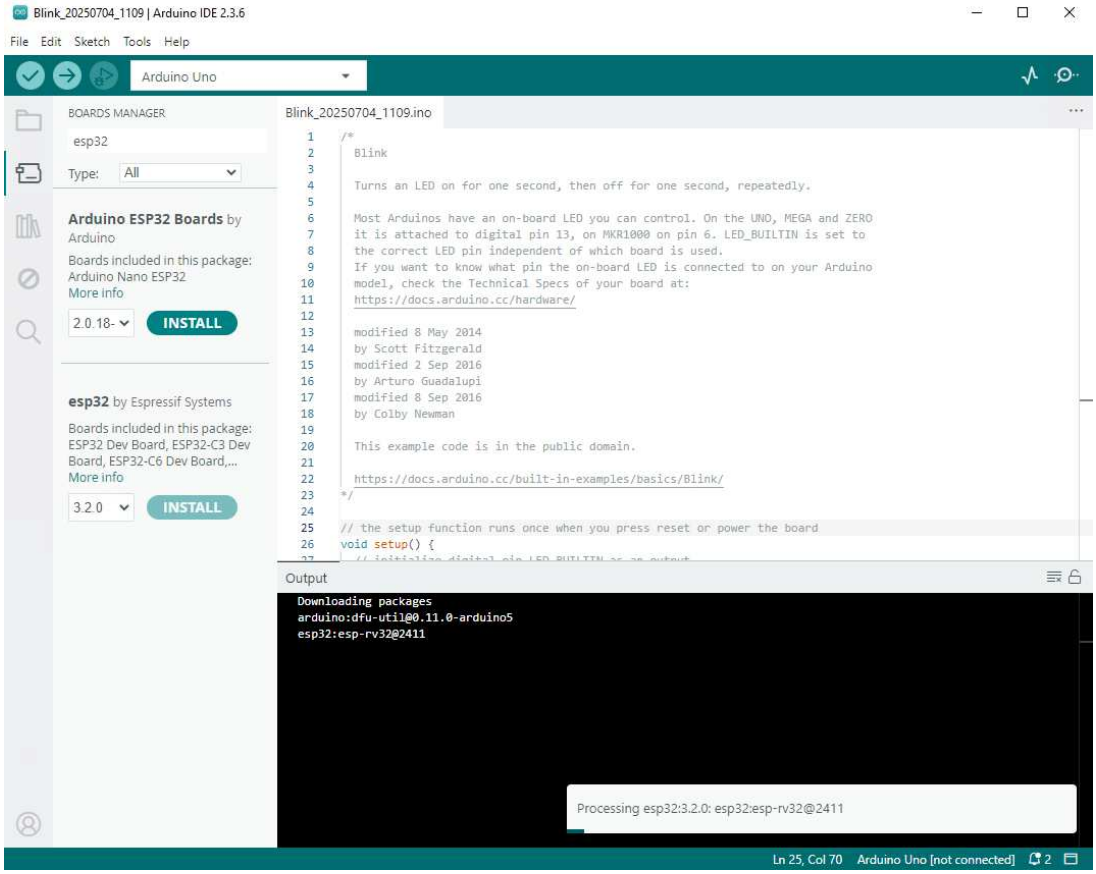
Środowiska Arduino IDE zaopatrzone w taką informacje wymaga aktywnego zainstalowania odpowiednich elementów – realizuje się to poprzez wybranie **`Tool -> Board`**:



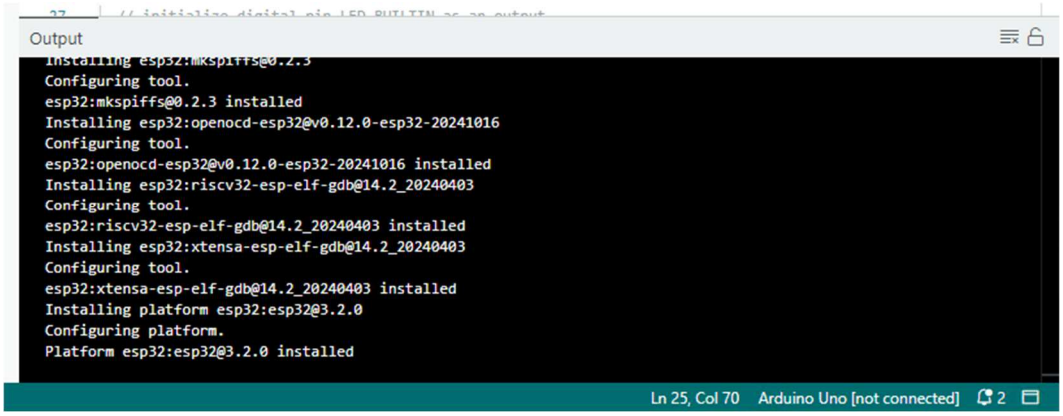
A następnie po pojawieniu się lewej wstążki wpisaniu ESP32:



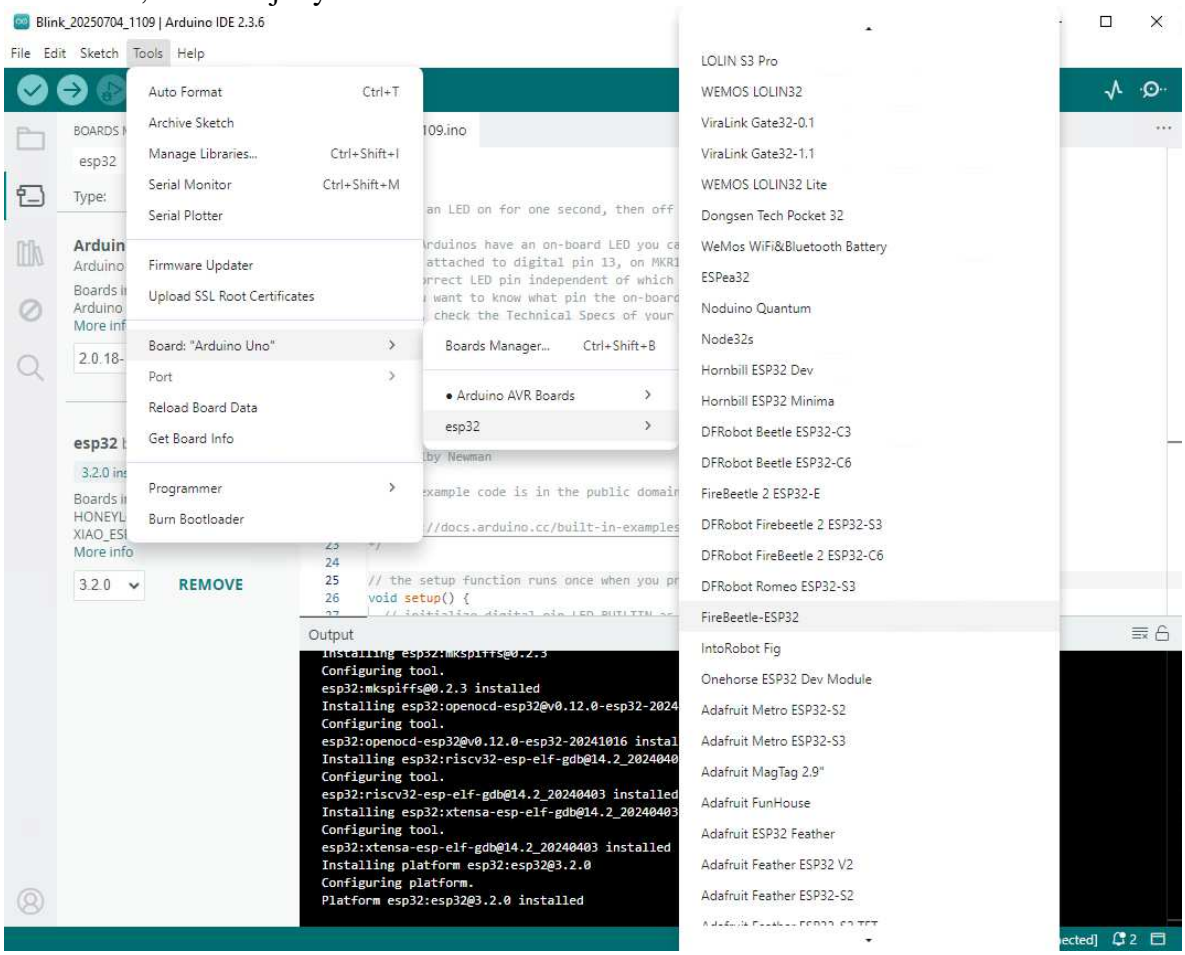
Z dwóch ukazanych opcji wybieramy „esp32 by Espressif Systems” klikając na klawisz „Install”. Po krótkiej chwili klawisz ten zmieni nieznacznie swój kolor:



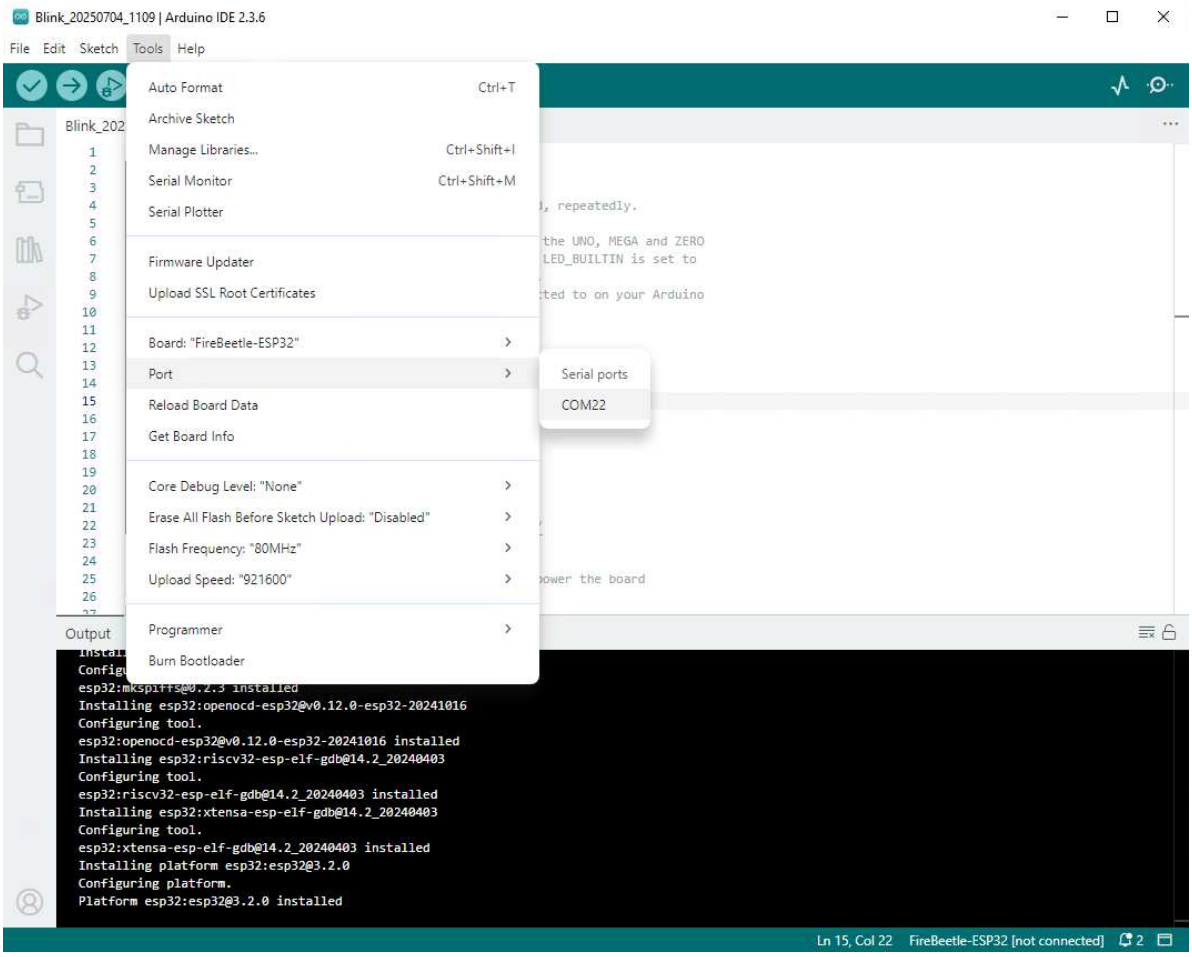
A system zacznie procedurę pobierania i instalacji odpowiednich narzędzi – proces ten może być dość czasochłonny. Gdy proces zakończy się powodzeniem w oknie „Output” ujrzymy:



Od tego momentu będziemy mogli używać w ramach zajęć laboratoryjnych płytek „FreeBettle Board-EPS32”, co ukazuje rysunek:



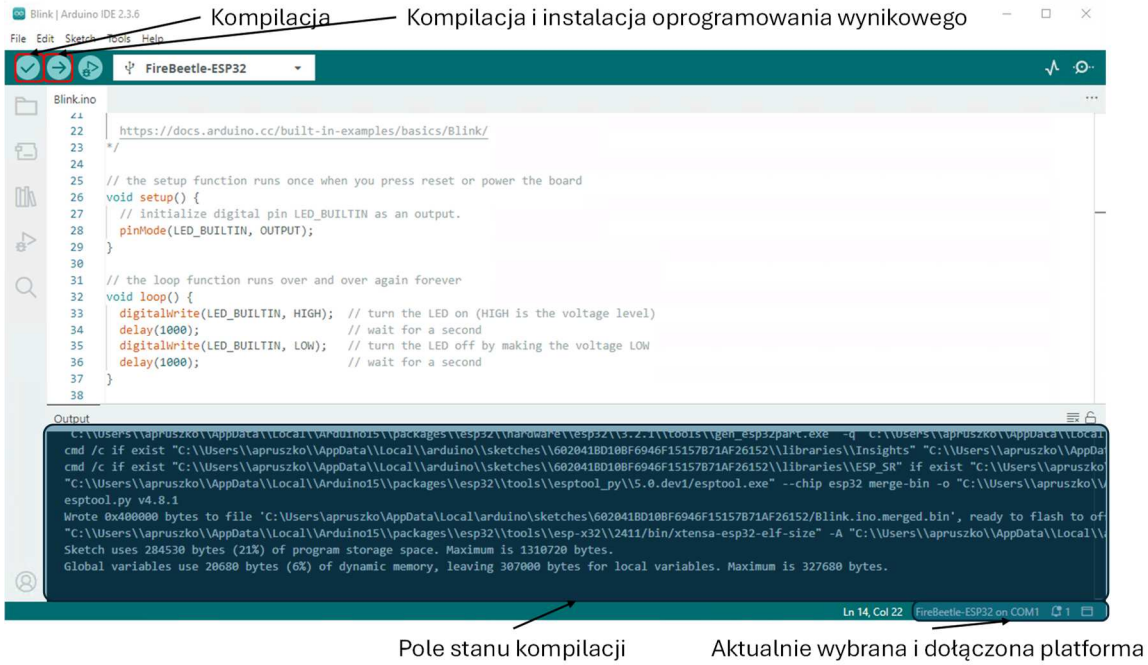
Płytkę „FreeBettle Board-EPS32” dołączamy poprzez kabel USB-microUSB do komputera a następnie w środowisku wybieramy port szeregowy pod jakim płytka będzie dostępna:



Jeśli pomimo powyższych zabiegów, nie będziemy mogli zlokalizować naszej płytki w spisie portów **`Tools->Port`**, może to oznaczać problem sprzętowy - proszę taki przypadek natychmiast zgłaszać prowadzącemu zajęcia.

B)Kompilacja i użytkowanie prostych programów przykładowych (ESP32, EBSimUnoEthCurses)

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 rysunek:



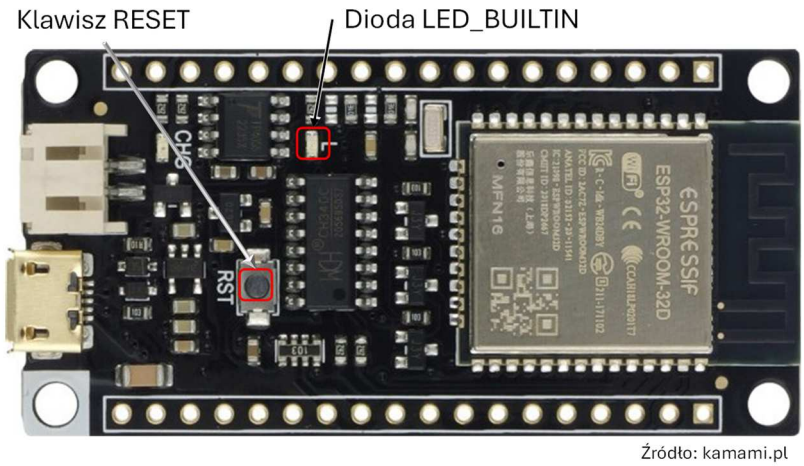
Po naciśnięciu przycisku opisanego ‘Kompilacja’, w polu stanu kompilacji prezentowany będzie przebieg tego procesu. Wszelkie błędy są także komunikowane w tym polu.
Przedstawiona tu aplikacja **`Blink`** (jak każda inna aplikacja w Arduino IDE) składa się ona z dwóch funkcji:

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

Pierwsza z tych funkcji jest wywoływana jednorazowo przez oprogramowanie systemowe zaraz po uruchomieniu i odpowiada za inicjalizację obsługiwanej platformy. Druga jest cyklicznie wywoływana przez cały czas działania platformy niemal zaraz po zakończeniu funkcji **`setup()`**.
UWAGA! Czas przez jaki procesor platformy wykonuje funkcję **`loop()`** powinien być jak najkrótszy, aby systemowe zadania ‘tła’ (np. obsługa niektórych interfejsów) mogły się właściwie wykonywać – system nie wspiera domyślnie wielowątkowości ani wielozadaniowości.
Po pomyślnej kompilacji i załadowaniu kodu do platformy ESP32, działanie aplikacji rozpoczyna się automatycznie. Gdyby proces ładowania zakończył się niepowodzeniem – proszę nacisnąć i przytrzymać klawisz RESET i równocześnie ponowić kompilację, puszczając klawisz RESET gdy w „Polu stanu kompilacji” pojawi się napis „Connecting.....”.
Aplikacja **`Blink`** w funkcji **`setup()`** konfiguruje pin GPIO podłączony do wbudowanej diody `LED_BUILTIN` jako wyjście. Natomiast funkcja **`loop()`** zmienia stan tego wyjścia GPIO, szybkość zmian wyznaczają wywołania funkcji **`delay()`**. Oto kod aplikacji Blink:

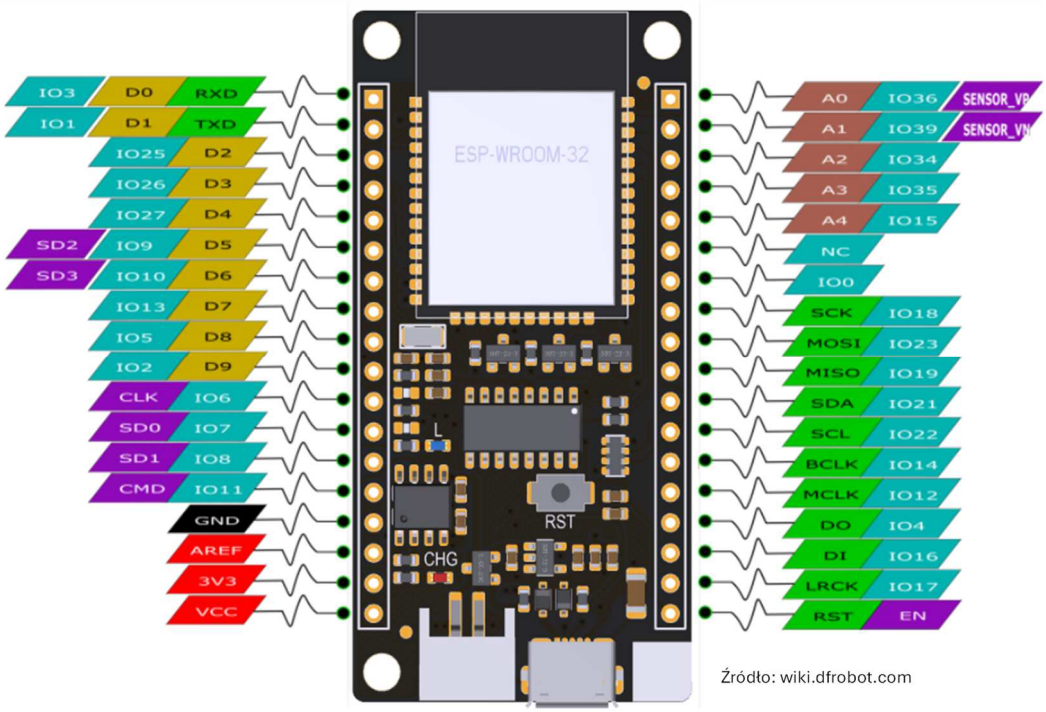
```
void setup() {
   pinMode(LED_BUILTIN, OUTPUT);      // initialize digital pin as an output.
}
void loop() {
   digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
   delay(1000);                       // wait for a second
   digitalWrite(LED_BUILTIN, LOW);    // turn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

Umiejscowienie diody LED podłączonej do wyjścia 13 pokazuje rysunek:



Źródło: kamami.pl

Dla dalszej pracy z platformą „FireBeetle Board-ESP32” może być ważne przyjęte przez producenta oznaczenie nazw styków, przedstawia je rysunek:



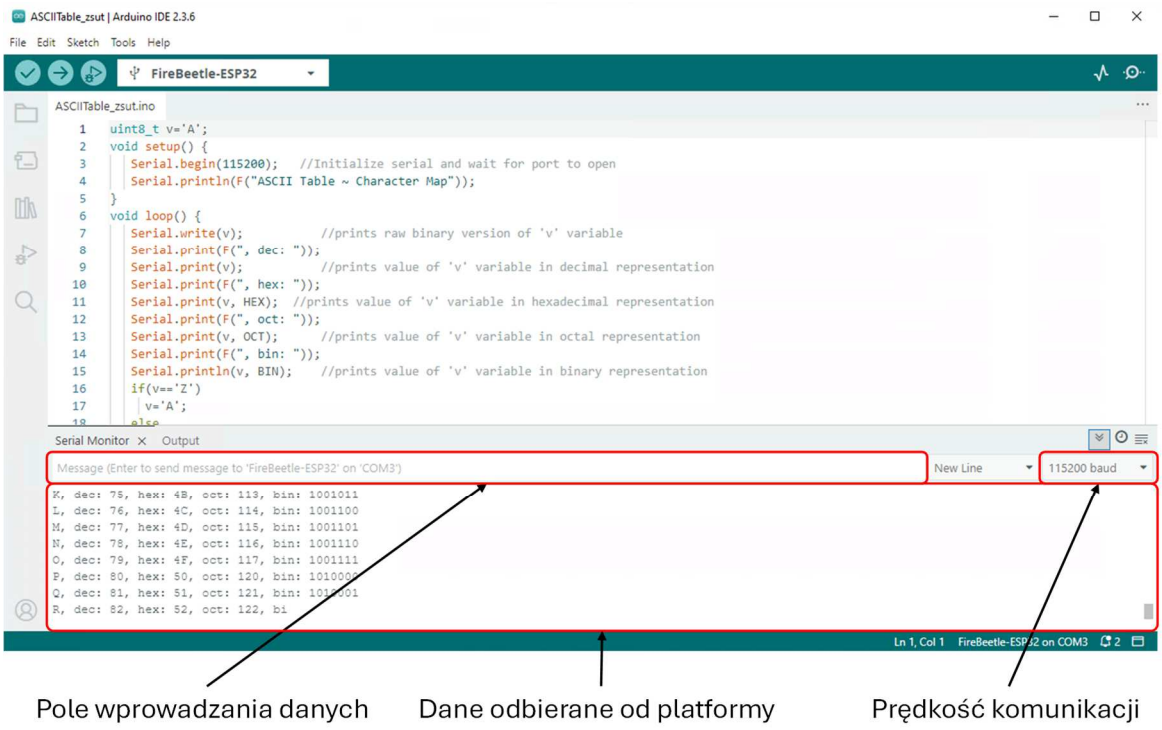
Drugim ważnym programem przykładowym jak postępować z peryferiami platformy pokazuje szkic o nazwie **`ASCIITable`**. Jest to prosty program obsługi portu szeregowego. Na listingu poniżej pokazano zmodyfikowaną wersję tego programu. Aplikacja ta stanowi test poprawności działania portu szeregowego. Dodatkowo, 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++;
}
```

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 rysunek:



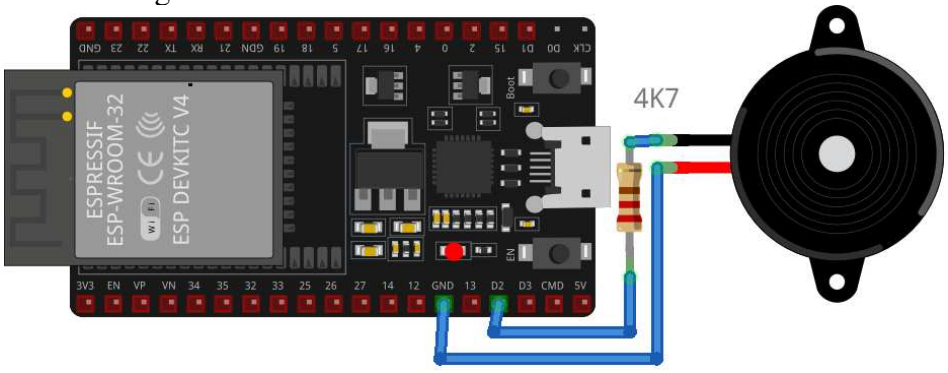
Nawiązując do wcześniejszego listingu, trzeba nadmienić, że w szkicach można używać dwóch wersji metody **`print()`**. I tak **`Serial.print()`** to zwykłe wypisanie treści za pomocą portu szeregowego, a **`Serial.println()`** działa niemal identycznie z tym że wypisana treść kończona jest znakiem/znakami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

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)Generowanie dźwięku (ESP32)

Podłączenie głośnika do FireBeetle Board-ESP32 pokazano na rysunku poniżej. 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.



Proste sterowanie głośnikiem pokazuje poniższy kod:

```
void setup() {
  pinMode(D2, OUTPUT);
}
void loop() {
  digitalWrite(D2, digitalRead(D2) ^ 1);   // toggle state of pin 2
  delay(1);
}
```

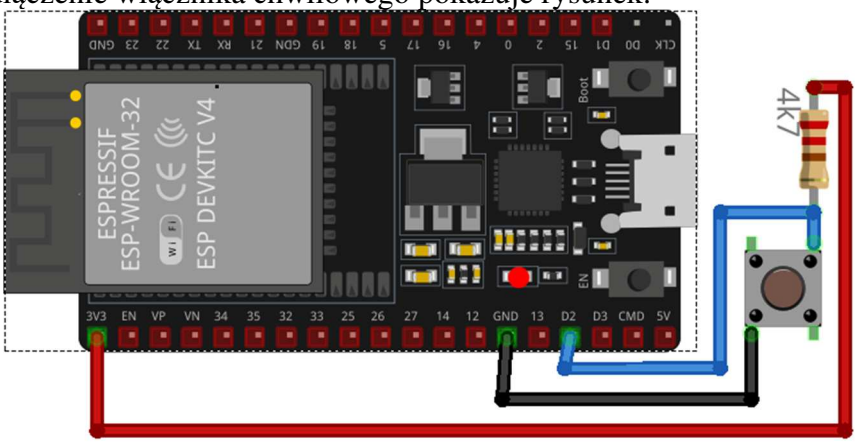
Należy zwrócić uwagę, że przedstawiony proces wymaga od procesora ciągłego sterowania membraną głośnika. Funkcja `digitalWrite()` jest odpowiedzialna za ruch membrany, zaś szybkość zmiany stanu pinu 2 wpływa bezpośrednio na ton generowanego dźwięku; w powyższym przykładzie będzie to około 500Hz. Sterowanie 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.
Metoda zastosowana w powyższym przykładzie nie jest jednak precyzyjna, i trudna w utrzymaniu gdyby kod był rozbudowywany. Jednym ze sposobów radzenia sobie z problemami tego typu jest zastosowanie liczników procesora wykorzystanego w ESP32. Licznik taki w połączeniu z mechanizmem przerwań może precyzyjnie odmierzać czas i uruchomiać procedurę obsługi przerwań co zadany interwał czasu a w niej może być zrealizowana obsługa zmiany stanu membrany głośnika. Przykład poniższy zbudowano na podstawie informacji i przykładów ze strony [`https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/api/timer.html`]:

```
hw_timer_t * timer = NULL;

void ARDUINO_ISR_ATTR myisr() {
  digitalWrite(D2, digitalRead(D2)^1); //actual tone generation
}
void setup() {
  pinMode(D2, OUTPUT);
  timer = timerBegin(1000000);          //Set timer frequency to 1Mhz
  timerAttachInterrupt(timer, &myisr); //attach callback - myisr routine
  timerAlarm(timer, 1000, true, 0);     //Set alarm to call onTimer function microseconds
                                        //(1000), and repeat the alarm (true), unlimited
                                        //count (0)
}
void loop() {
}
```

D)Obsługa włącznika chwilowego (ESP32)

Przykładowe podłączenie włącznika chwilowego pokazuje rysunek:



W programie można sprawdzić stan takiego przełącznika dość prosto:

```
void setup() {
  pinMode(D2, INPUT);
}
void loop() {
  if (digitalRead(D2) == 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://docs.arduino.cc/built-in-examples/digital/Debounce/
```

Wprowadzenie w specyficzne aspekty związane z korzystaniem z platformy emulowanej

A)Platforma emulowana (EBSimUnoEthCurses)

Dla potrzeb tego ćwiczenia laboratoryjnego została także przygotowana – podobnie jak dla ćwiczenia 1, maszyna wirtualna jest 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 CLI. 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.

Prace z wykorzystaniem symulatora **EBSimUnoEthCurses** dość wiernie odwzorowują działanie sprzętowej platformy Arduino UNO z nakładką Ethernet. **EBSimUnoEthCurses** jest to program, który emuluje MCU zainstalowane w Arduino UNO czyli Atmega328P na poziomie pojedynczych instrukcji procesora wbudowanego w to MCU. Dla potrzeb dalszej części teksu można zdefiniować pojęcie „platforma emulowana" (czyli działające oprogramowanie **EBSimUnoEthCurses**) 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()`.
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

[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.

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)Przygotowanie środowiska Arduino IDE (EBSimUnoEthCurses)

Dla prac związanych z tworzeniem kodu wymagane jest wybranie w Arduino IDE platformy Arduino UNO. W miejsce tego IDE można użyć środowiska Platformio Core (do pobrania ze strony: https://docs.platformio.org/en/latest//core/installation.html).
UWAGA! Instalacji Arduino IDE nie wykonuje się na maszynie wirtualnej `psir23z_20230908_1052.ova` a na komputerze goszczącym tę maszynę wirtualną (lub jakimkolwiek będącym pod naszą kontrolą, w laboratorium komputery posiadają zainstalowane już to IDE).

Po uruchomieniu Arduino IDE przed pierwszymi pracami należy skonfigurować i dodać biblioteki dla **EBSimUnoEthCurses**. Odpowiednia paczka oprogramowania jest umieszczana na serwerze Studia. Paczka ta zawiera katalog `ZsutEthernet` który należy skopiować (wraz z podkatalogami) do katalogu:

`C:\Users\zsutguest\Documents\Arduino\libraries`

Powyższe miejsce jest domyślnym miejscem składowania bibliotek na komputerze Windows PC gdzie pracuje użytkownik `zsutguest`. W razie pracy jako inny niż `zsutguest` użytkownik konieczne jest odpowiednia modyfikacja tego miejsca. Program Arduino IDE w swoich `Preferences` podaje umiejscowienie katalogu gdzie będzie składował szkice (programy źródłowe tworzone przez użytkownika) a środowisko domyślnie umieszcza w tym samym katalogu pobrane i zainstalowane biblioteki.

C)Instalacja środowiska Platformio Core (EBSimUnoEthCurses)

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.

D)Kompilacja aplikacji użytkownika w środowisku Arduino IDE (EBSimUnoEthCurses)

Dla celów dydaktycznych prowadzący zajęcia laboratoryjne udostępnia także dwa pliki w postaci źródłowej: `ZsutEthUdpServer.cpp i ZsutUdpNtpClient.cpp`. Stanowią one pomoc w testowaniu zestawionego środowiska, jak i w tworzeniu nowego kodu i są one także dostępne w paczce z plikami bibliotecznymi na serwerze Studia.

Po napisaniu kodu aplikacji użytkowej (ustalmy, że będzie to szkic który zapiszemy na komputerze korzystając z opcji „`File`”-> „`Save As`” pod nazwą: `ZsutUdpNtpClient`) można z użyciem Arduino IDE utworzyć plik z obrazem pamięci kodu (tzw. firmware) dla emulatora **EBSimUnoEthCurses.** Aby to wykonać trzeba plik z kodem aplikacji użytkowej umieścić w okienku głównym Arduino IDE a następnie wybrać „`Narzędzia`” -> „`Płytka`” -> „`Arduino Uno`” i dokonać kompilacji „`Szkic`” -> „`Weryfikuj`” (ang. Verify). Podejście takie jest domyślnym posługiwaniem się środowiskiem, jednakże dla wygody współpracy z **EBSimUnoEthCurses** można skorzystać z niedawno wprowadzonej usługi dostępnej poprzez menu programu wybierając „`Sketch`”->”`Export Compiled Binary`”. Niestety skrót klawiszowy ALT+CTRL+s nie działa poprawnie z polskimi klawiaturami.

Usługa ta tworzy w katalogu gdzie przechowywany jest szkic katalog `build` a w nim katalog `arduino.avr.uno` w którym przechowywane będą wyniki kompilacji dla wcześniej ustalonej nazwy szkica będzie to `ZsutUdpNtpClient.ino.hex`.

E)Kompilacja w środowisku Platformio (EBSimUnoEthCurses)

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`

F)Uruchomienie kompilatu przez emulator (EBSimUnoEthCurses)

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:

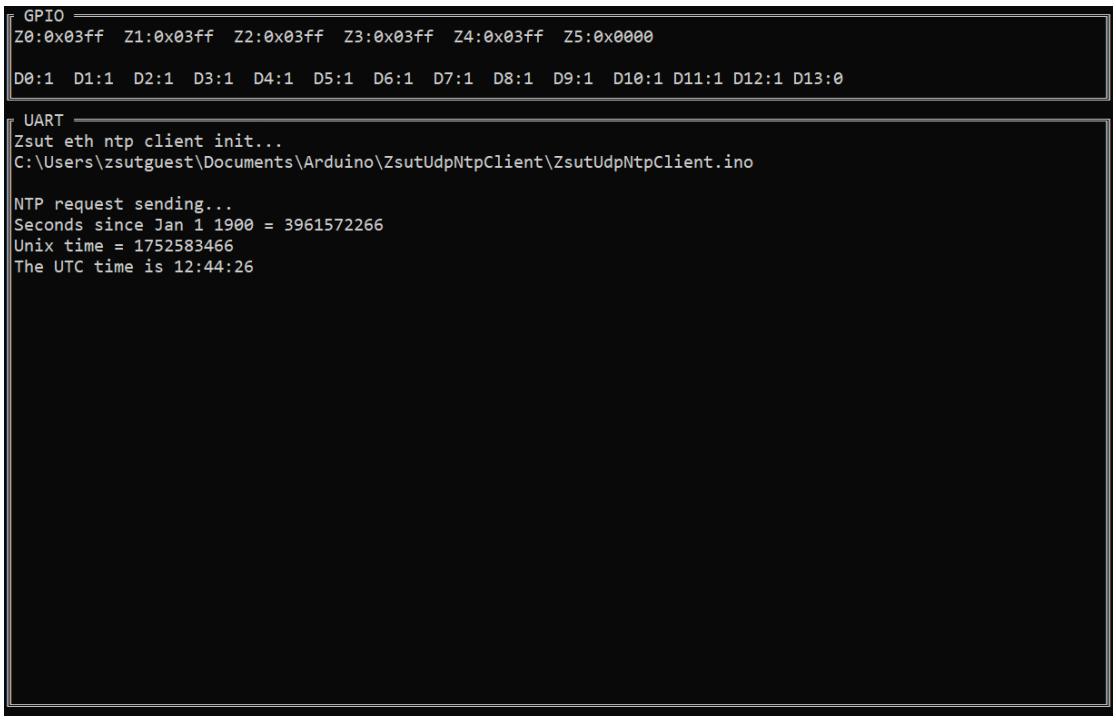
```
EBSimUnoEthCurses –ip 192.168.89.3 ZsutUdpServer.hex
```

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

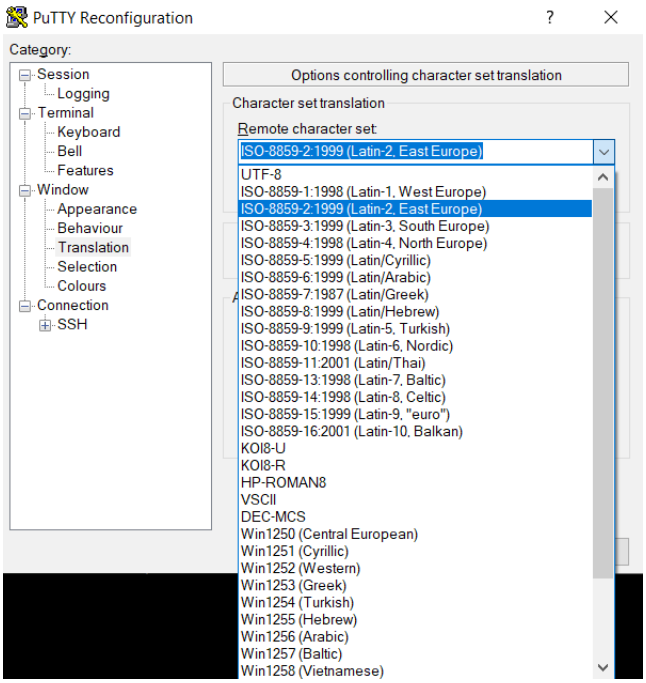
```
./EBSimUnoEthCurses –ip 192.168.89.3 ZsutUdpServer.hex
```

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 tzw. rysowania brzydkich ramek w niektórych środowiskach w których używany jest **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:



Proszę pamiętać, że po uruchomieniu program **EBSimUnoEthCurses** w nieskończoność emuluje działanie węzła **Arduino UNO**, aby jednak możliwe było przerwanie jego pracy pozostawiono możliwość skorzystania z 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.
W obecnej wersji 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 (ESP32, EBSimUnoEthCurses)

Aby zapewnić pełna komunikacje węzłów z wykorzystaniem sieci Internet należy stosować się do zaleceń związanych z modelem programistycznym Arduino. Jest on nieznacznie rózny od systemu POSIX.

EBSimUnoEthCurses emuluje Arduino UNO z nakładką Ethernet a ta 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:

a)Dołączyć odpowiednie pliki nagłówkowe:

ESP32	EBSimUnoEthCurses

`#include <WiFi.h>`
`#include <WiFiUdp.h>`
	`#include <ZsutEthernet.h>`
	`#include <ZsutEthernetUdp.h>`
	`byte mac[]={`
	`  0x01, 0xff, 0xaa, 0xXX, 0xYY, 0xZZ`
	`};`

b)wyłącznie dla platformy emulującej (ESP32 ma fabrycznie zapisany numer MAC), określić adres MAC (typowo programy raportujące ten numer podają go jako np.: 01:ff:aa:xx:yy:zz – prowadzący zajęcia podaje właściwy numer MAC każdej grupie) swojego urządzenia w 6 bajtowej tablicy `mac`, tutaj dla emulatora to:

ESP32	EBSimUnoEthCurses

	`byte mac[]={`
	`  0x01, 0xff, 0xaa, 0xXX, 0xYY, 0xZZ`
	`};`

Mimo, 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 pamiętać.

c)w przypadku ESP32 należy zdefiniować z jakim punktem dostępowym (tj. ssid) nasz węzeł ma pracować oraz zdefiniować hasło dostępowe (tj. pwd) dla tego punktu (dla **EBSimUnoEthCurses** nie stosuje się tych informacji – w tym węźle komunikacja jest emulowana za pomocą systemu Linux):

ESP32	EBSimUnoEthCurses
`const char * ssid = "dd-wrt";` `const char * pwd =  "00000000";`	

d)utworzyć obiekt (globalny) typu dla obsługi połączeń UDP:

ESP32	EBSimUnoEthCurses
`WiFiUDP 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:

ESP32	EBSimUnoEthCurses

`WiFi.`**`begin`**`(ssid, pwd);`
`while(WiFi.`**`status`**`() != WL_CONNECTED){`
`  delay(500);Serial.`**`print`**`(".");`
`}`
`Serial.`**`println`**`(WiFi.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:

ESP32

```
#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.

4. Literatura

1.Kodowanie ASCII https://en.wikipedia.org/wiki/ASCII, ostatnia wizyta 2025.11.02