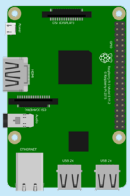
Android Things oraz Raspberry Pi 3



Obsługa magistrali I²C

W kolejnej części cyklu poświęconej systemowi Android Things oraz płytce Raspberry Pi, na przykładzie graficznego wyświetlacza OLED z kontrolerem SSD1306 oraz układu HTU21 będącego zintegrowanym modułem do pomiaru temperatury i wilgotności względnej powietrza, przedstawiona zostanie programowa obsługa interfejsu I²C. Aby lepiej zaprezentować korzyści wynikające z wykorzystania systemu operacyjnego Android Things i dostępnych dla systemu Android bibliotek programowych, przykład zostanie rozszerzony o interfejs graficzny, prezentujący uzyskane wyniki pomiarów w formie czytelnych wykresów. Zaczynajmy!

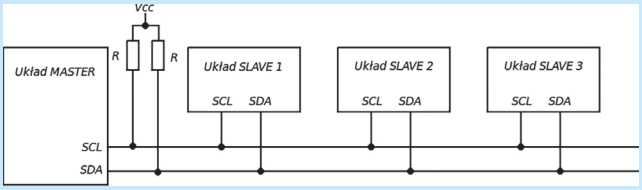
Współczesny rynek elektroniki użytkowej wymaga od projektantów i konstruktorów, by projektowane przez nich urządzenia były nie tylko funkcjonalne i poprawnie wykonane, ale również, by czas od fazy wczesnego prototypu do wdrożenia gotowego produktu na rynek był możliwie krótki, a pierwszy prototyp – tym ostatnim. Dlatego też, wielu producentów sprzętu w swoich rozwiązaniach decyduje się na wykorzystanie bardziej zasobnych sprzętowo mikrokontrolerów i mikroprocesorów, umożliwiających uruchomienie pełnego systemu operacyjnego, takiego jak np. Linux czy Android.

Zwiększony nakład finansowy na warstwę sprzętową jest rekompensowany poprzez skrócenie czasu pracy nad oprogramowaniem i jego testami. Dzięki wykorzystaniu systemu operacyjnego, programista urządzenia uzyskuje dostęp do ogromnych zasobów gotowych komponentów programowych (np. w postaci bibliotek), przetestowanych przez miliony użytkowników. Co więcej, system operacyjny pełni rolę wydajnego menadżera zasobów oraz zapewnia abstrakcję warstwy sprzętowej, dzięki czemu programista nie musi znać niskopoziomowej specyfiki podłączonych urządzeń peryferyjnych.

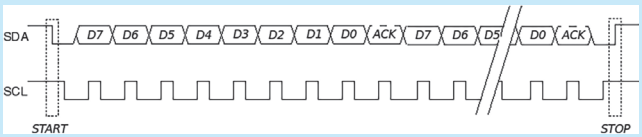
Kilka słów o I²C

Interfejs I²C jest popularnym i powszechnie wykorzystywanym w systemach wbudowanych, synchronicznym interfejsem szeregowym. Mając jednak na uwadze fakt, że część Czytelników niniejszego artykułu może „wywodzić się" z grupy programistów systemu *Android* i urządzeń mobilnych, a dzięki *Android Things* dopiero rozpoczyna swoje przygody z elektroniką, poniżej przedstawiono zwięzły opis teoretycznych podstaw działania magistrali I²C.

Magistrala I²C jest dwukierunkowym i dwuprzewodowym interfejsem szeregowym, przeznaczonym do komunikacji pomiędzy układem nadrzędnym (mikrokontrolerem), a innymi układami peryferyjnymi umieszczonymi w obrębie jednego systemu, na przykład: czujnikami temperatury, zegarami czasu rzeczywistego, przetwornikami C/A oraz A/C. Transmisja na magistrali realizowana jest w sposób synchroniczny z wykorzystaniem dwóch linii: *SDA* (linia danych) oraz *SCL* (linia zegarowa). Za poprawną pracę magistrali I²C odpowiada układ nadrzędny *Master*, do którego zadań należy generowanie sygnału zegarowego na linii *SCL* i synchronizacja wymiany danych. Linie interfejsu I²C są liniami typu otwarty dren, co oznacza, że magistrala do poprawnej pracy wymaga zewnętrznych rezystorów „podciągających". Rezystory



Rysunek 1. Schematyczna budowa magistrali I²C



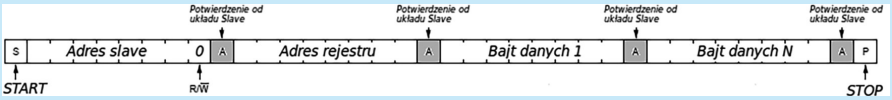
Rysunek 2. Transmisja danych na magistrali I²C na poziomie bitowym

te są odpowiedzialne za ustawienie stanu wysokiego na magistrali w przypadku braku transmisji. Standard magistrali I²C umożliwia podłączenie wielu układów podrzędnych (układów typu *Slave*), co narzuca konieczność wprowadzenia odpowiedniego mechanizmu adresowania. Układy *Slave* są adresowane poprzez 7-bitowy adres – ostatni 8. bit określa wówczas kierunek przepływu kolejnej transakcji danych (odczyt/zapis). Sam standard magistrali udostępnia również możliwość wykorzystania 10-bitowego adresowania układów, jednak wariant ten nie będzie omawiany w ramach niniejszego artykułu. Schematyczna budowa magistrali I²C została przedstawiona na **rysunku 1**.

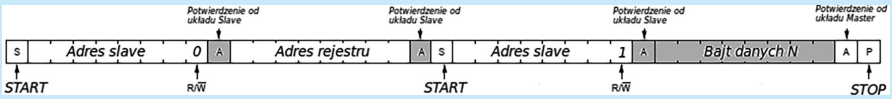
Dane przesyłane za pomocą interfejsu I²C grupowane są w 8-bitowe bloki. Rozpoczęcie transmisji odbywa się po wystawieniu przez układ *Master* sekwencji *START*, która identyfikuje początek ramki oraz rezerwuje magistralę, aż do wystąpienia sekwencji *STOP*. Przy sekwencji startowej, układ *Master* ściąga linię *SDA* do poziomu niskiego, przy wysokim stanie linii zegarowej. Koniec transmisji ramki jest operacją odwrotną – zbocze narastające na linii *SDA* przy wysokim poziomie zegara. Sygnał startu przełącza wszystkie podpięte do magistrali układy w tryb odbioru danych. Od tego momentu dane przesyłane za pomocą linii *SDA* są ważne i mogą zmieniać się wyłącznie w stanie niskim linii zegarowej. Po ośmiu taktach zegara (nadaniu bajtu danych) zaadresowany układ *Slave*, musi potwierdzić prawidłowość odbioru przesyłanych informacji. Czynność ta realizowana jest poprzez bit *ACK* (ang. *ACKnowledge* – potwierdzać), czyli wymuszenie przez układ *Slave* stanu niskiego na linii *SDA*. Układy podrzędne mają również możliwość „ściągnięcia do masy" sygnału zegarowego

w wypadku, gdy nie nadążają z odbiorem danych lub realizują inne zadania (np. realizują na żądanie układu *Master* proces pomiaru temperatury lub wilgotności). Typowy przebieg transmisji na poziomie bitowym przedstawiono na **rysunku 2**.

Na **rysunku 3** oraz **rysunku 4** przedstawiono dwa typowe scenariusze przebiegu transmisji na poziomie bajtowym (typowe dla układów *Slave* posiadających organizację rejestrową, tzn. gdzie komunikacja z układem odbywa się poprzez zapis/odczyt określonych rejestrów urządzenia pełniących przypisane przez producenta funkcje).



Rysunek 3. Przebieg transmisji na poziomie bajtowym – zapis danych do układu Slave



Rysunek 4. Przebieg transmisji na poziomie bajtowym – odczyt danych z układu Slave

W pierwszym z przypadków (rysunek 3) układ Master generuje adres układu podrzędnego z ósmym bitem o wartości 0 – zapis do układu *Slave*. Po wygenerowaniu adresu, wybrany układ podrzędny odpowiada bitem potwierdzenia *ACK* (o ile istnieje urządzenie Slave o zadanym adresie). Kolejny bajt zapisywany do urządzenia *Slave* jest numerem rejestru, do którego *Master* będzie zapisywał dane. Zapis danych, będzie realizowany do momentu wygenerowania sekwencji *STOP*. Procedura odczytu zorganizowana jest w analogiczny sposób – po zaadresowaniu układu podrzędnego i wyborze rejestru z którego będziemy dokonywać odczytu, kolejne bajty danych generowane są przez układ *Slave* – układ *Master* dokonuje odczytu i potwierdza ich odbiór bitem *ACK*.

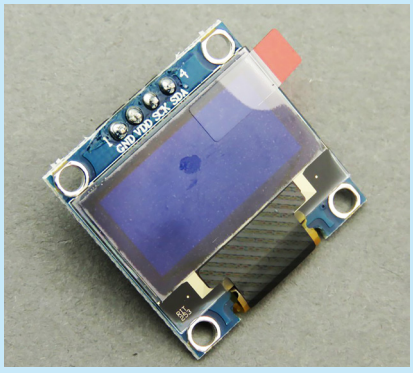
Warstwa sprzętowa projektu AndroidThings-I2C

Obsługa magistrali I²C w systemie *Android Things* zostanie przedstawiona na przykładzie dwóch układów peryferyjnych:

- Modułu *modOLED096_I2C* wyposażonego w miniaturowy wyświetlacz OLED z wbudowanym sterownikiem *SSD1306* (**rysunek 5**).
- Modułu *modHTU21* z czujnikiem wilgotności i temperatury *HTU21* (**rysunek 6**).

W oparciu o wyżej wymieniony moduły, przygotowana zostanie prosta aplikacja *AndroidThings-I2C*, realizująca pomiar temperatury i ciśnienia w 1-sekundowych odstępach czasu. Uzyskane wyniki pomiarów będą prezentowane w postaci wykresów na ekranie podłączonym do wyjścia HDMI zestawu *Raspberry Pi*. Aby zmaksymalizować walor dydaktyczny, w realizowanym projekcie przyjęto następujące założenia:

- Obsługa modułu wyświetlacza *SSD1306* zostanie zrealizowana z wykorzystaniem gotowych sterowników udostępnianych przez system *Android Things* i rozwijaną wraz z systemem bibliotekę *Peripheral Driver Library* [1].
- Do obsługi modułu *modHTU21* zostaną wykorzystane funkcje bezpośredniej obsługi interfejsu I²C z biblioteki *Things Support Library*.
- Interfejs graficzny aplikacji zostanie zbudowany w oparciu o bibliotekę *MPAndroidChart*, tak aby zaprezentować łatwość wykorzystania otwartoźródłowych, zewnętrznych bibliotek rozwijanych aktywnie przez ogromną społeczność programistów systemu *Android*.



Rysunek 5. Moduł modOLED096_I2C z wyświetlaczem OLED i budowanym sterownikiem SSD1306

W typowych projektach *embedded* budowanych „od podstaw" i bez wykorzystania systemu operacyjnego, niezbędnym krokiem jest dokładne zapoznanie się z dokumentacją i programową obsługą wszystkich podłączonych komponentów (konfiguracją urządzenia, układem rejestrów, formatem wymiany danych, itp.). W zależności od złożoności układu (a tym samym obszerności jego dokumentacji), etap ten może być bardzo czasochłonny. Wykorzystanie rozbudowanych systemów operacyjnych oraz sterowników sprzętu udostępnianych przez te systemy, pozwala maksymalnie skrócić czas przygotowania warstwy programowej. Jak zostanie to przedstawione na przykładzie wyświetlacza z kontrolerem *SSD1306*, dzięki wykorzystaniu gotowego sterownika i czytelnego zestawu udostępnianych przez niego funkcji, programista może uniknąć żmudnego etapu studiowania dokumentacji i przygotowywania niskopoziomowej obsługi wyświetlacza. Aby jednak nie pominąć istotnych zagadnień związanych z obsługą magistrali I²C w systemie *Android Things*, obsługa czujnika *HTU21* zostanie zrealizowana z wykorzystaniem „niskopoziomowych" funkcji zarządzających pracą magistrali (niskopoziomowych w sensie funkcji realizujących odczyt i zapis danych na magistrali, a nie obsługi sprzętowego kontrolera I²C poprzez zapis/odczyt rejestrów mikroprocesora).

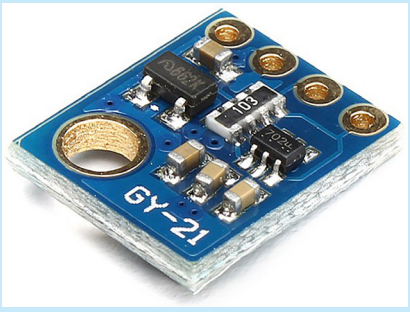
Komplet połączeń sprzętowych pomiędzy płytką deweloperską *Raspberry Pi* a modułami *modOLED096_I2C* oraz *modHTU21*, przedstawiono na **rysunku 7**.

Warstwa programowa projektu AndroidThings-I2C

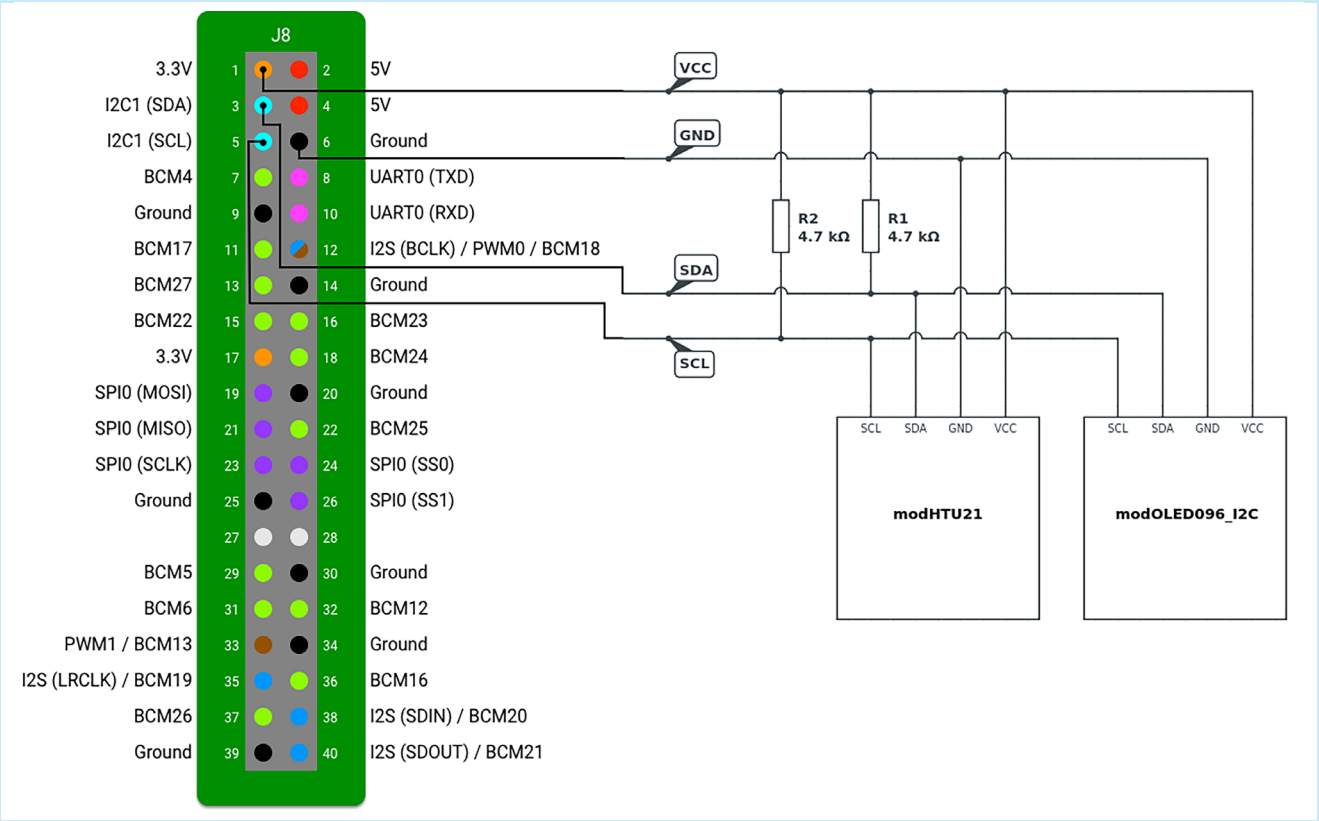
Korzystając z pakietu *Android Studio*, jak i informacji przedstawionych w poprzedniej części cyklu [**LINK**], utwórzmy nowy projekt – *AndroidThings-I2C*. Jeżeli nie korzystamy z przedpremierowej wersji *Android Studio* (oznaczonej numerem 3.0), należy pamiętać, aby dokonać odpowiednich modyfikacji plików *build.gradle* oraz *AndroidManifest.xml*, na potrzeby systemu *Android Things*.

Do budowy graficznego interfejsu użytkownika zostanie wykorzystana otwartoźródłowa biblioteka *MPAndoridChart*, udostępniona na licencji *Apache 2.0*, pod adresem ***https://goo.gl/v5BcKT***.

Aplikacje tworzone w środowisku *Android Studio* korzystają z wydajnego i przystępnego dla użytkownika narzędzie *Gradle* [2] (zarządzającego procesem budowy projektu), dzięki czemu włączenie w projekt zewnętrznych bibliotek ogranicza się do edycji kilku linii skryptów budujących. Konfigurację zależności dla biblioteki *MPAndroidChart*, rozpoczniemy od edycji piku *build.*



Rysunek 6. Moduł modHTU21 z czujnikiem wilgotności i temperatury HTU21



Rysunek 7. Schemat podłączenie modułów modHTU21 oraz modOLED096_I2C

gradle, gdzie w sekcji *repositories*, dodamy repozytorium ***https://jitpack.io***, w którym to skrypt budujący będzie poszukiwał bibliotek wymaganych dla etapu kompilacji:

```
allprojects {
    repositories {
        jcenter()
        maven { url „https://jitpack.io” }
    }
}
```

Po określeniu repozytorium, możemy przystąpić do edycji pliku *app/build.gradle*, gdzie w sekcji *dependencies*, zdefiniujemy właściwe zależności:

```
dependencies {
    /.../
   compile ‚com.github.PhilJay:MPAndroidChart:v3.0.2’
    /.../
}
```

W projekcie *AndroiThings-I2C*, oprócz biblioteki *MPAndroidChart*, wykorzystana zostanie również biblioteka do obsługi wyświetlacza *SSD1306*. Aby usprawnić proces tworzenia aplikacji, deweloperzy z firmy Google, wraz z systemem *Android Things* rozwijają projekt biblioteki *Peripheral Driver Library*, która implementuje niskopoziomową obsługę najbardziej popularnych układów peryferyjnych. Kod źródłowy biblioteki wraz z gotowymi przykładami jej obsługi, został udostępniony pod adresami:

https://goo.gl/9fUPsM
https://goo.gl/CAvD2P

Włączenie sterownika *SSD1306* z biblioteki *Peripheral Driver Library* wymaga ponownej edycji pliku *app/build.gradle* i dodania w sekcji *dependencies* nowej zależności, jak niżej:

```
dependencies {
    /.../
   compile ‚com.github.PhilJay:MPAndroidChart:v3.0.2’
    /.../
}
```

Przed implementacją głównej funkcjonalności projektu, przystąpmy do edycji pliku *layout/activity_main.xml* i zdefiniowania prostego graficznego interfejsu użytkownika. Do budowy interfejsu wykorzystamy układ *LinearLayout* [3] z atrybutem *Vertical*, co oznacza, że wszystkie zdefiniowane elementy interfejsu, będą umieszczone kolejno od góry do dołu ekranu. Interfejs aplikacji będzie złożony z czterech komponentów: napisu wyświetlającego aktualną wartość temperatury (jako komponent *TextView* z bibliotek systemu Android), wykresu przedstawiającego wyniki ostatnich 60. pomiarów temperatury (wykres typu *LineChart* z biblioteki *MPAndroidChart*), napisu wyświetlającego aktualną wartość wilgotności względnej oraz wykresu jak dla pomiarów temperatury. Schematyczny rozkład komponentów został przedstawiony na **rysunku 8**.

Wszystkim komponentom interfejsu przypisujemy unikalną wartość pola ID (*android:id*), tak aby mieć możliwość do bezpośredniego odwołania i programowej zmiany ich parametrów z poziomu klasy *MainActivity*. Dla komponentów typu *TextView* ustawiamy pożądane wartości koloru (*android:textColor*), stylu (*android:textStyle*)

Rysunek 8. Schematyczny rozkład komponentów dla pliku layout/activity_main.xml

i rozmiaru czcionki (*android:textSize*), koloru tła (*android:background*), a także odległości dzielących je od pozostałych komponentów interfejsu (*android:padding*). Wizualne aspekty komponentów *LineChart* zostaną zdefiniowane bezpośrednio z poziomu klasy *MainActivity*. Kompletną postać pliku *layout/activity_main.xml*, przedstawiono na **listingu 1**.

Zatem przystąpmy do finalnej implementacji klasy *MainActivity*. Tworzenie aplikacji rozpoczniemy od obsługi czujnika *HTU21*. Do tego celu wykorzystamy niskopoziomowe funkcje obsługi interfejsu I²C oraz znany już z poprzedniego odcinka *PeripheralManagerService*. Poprzez metodę *openI2cdevice()* i określenie 7-bitowego adresu urządzenia *Slave* (w przypadku modułu *HTU21*, adres ma stałą, niekonfigurowalną wartość *0x40*), uzyskajmy dostęp do magistrali I²C, jak na **listingu 2**.

Klasa *I2cDevice* [4] udostępnia szereg niskopoziomowych metod, umożliwiających wygodną obsługę magistrali. Wybrane metody klasy *I2cDevice*, zostały przedstawione w **tabeli 1**.

Metody z grupy *writeReg*()* oraz *readReg*()* pozwalają na wygodną obsługą urządzeń peryferyjnych posiadających organizację rejestrową (w której odczyt/zapis danych jest poprzedzony wskazaniem adresu urządzenia). Za pomocą tych pojedynczych metod realizujemy cały cykl działań na magistrali I²C (przesłanie adresu rejestru, adresu urządzenia *Slave* z odpowiednio ustawionym bitem kierunku przesyłu informacji, itd.), jak został to przedstawione na *rysunku 3* oraz *rysunku 4*. Ponieważ moduł *HTU21* nie posiada organizacji rejestrowej, do wymiany danych z modułem wykorzystamy funkcje *write()* oraz *read()*, które pozwalają nam uzyskać pełną kontrolę na danymi przesyłanymi na magistrali I²C.

System *Android Things* nie zawiera dedykowanego sterownika dla układu *HTU21*, więc JEST niezbędne zapoznanie się z dokumentacją układu i jego programową obsługą [5]. Lista komend obsługiwanych przez czujnik *HTU21* została przedstawiona w **tabeli 2**. Na podstawie funkcji z tab. 1 oraz listy komend z tab. 2 przygotujmy generyczną funkcję, której zadaniem będzie pomiar zadanej wartości mierzonej w trybie „*Hold Master*", a także sprawdzenie sumy *CRC* z wielomianem generującym $X^8+X^5+X^4+1$ (*CRC-8-Dallas*). Kod funkcji *htu21_readData()* został przedstawiony na **listingu 3**.

Pomiar temperatury i wilgotności będzie realizowany w 1-sekundowych odstępach czasu. Do zapewnienia cykliczności pomiarów wykorzystamy mechanizm *Runnable*, który wykorzystany został również w implementacji „migającej diody" w pierwszej części cyklu poświęconej systemowi *Android Things*. Aby upewnić się, że moduł *HTU21*, do którego uzyskaliśmy dostęp we fragmencie kodu z listingu 2, odpowiada na przesyłane zapytania, przed uruchomieniem mechanizmu *Runnable*, prześlijmy komendę zerującą układ – *SoftReset* (0xFE). W tym celu rozbudujemy metodę *onCreate()* o fragment kodu pokazany na **listingu 4**. Dla pełnej czytelności kodu, warto przedstawić również implementację funkcji *htu21_softReset()* – pokazano ją na **listingu 5**.

Zgodnie z przyjętymi założeniami, wyzwolenie pomiarów temperatury i ciśnienia oraz aktualizacja graficznego interfejsu użytkownika będzie realizowana w 1-sekundowych odstępach czasu. Do zadań funkcji wyzwalających pomiar, będzie należało również odpowiednie przeliczenie wartości pomiarowych, przesłanych przez moduł *HTU21*. Dokumentacja modułu [5] definiuje dwa wzory, pozwalające na uzyskanie rzeczywistych wartości wielkości mierzonej, wyrażonej w stopniach Celsjusza (dla temperatury) oraz w procentach (dla wilgotności względnej):

$$Temp=-46{,}85+175.72\times(S_{Temp}/2^{16})$$

$$RH=-6+125\times(S_{RH}/2^{16})$$

gdzie:

- S_{Temp} – 16-bitowa wartość odczytana z modułu po przesłaniu komendy 0xE3 lub 0xF3.

Listing 1. Opis interfejsu użytkownika w pliku layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/temp_textView"
        android:textColor="@android:color/white"
        android:textSize="40sp"
        android:textStyle="bold"
        android:padding="10sp"
        android:background="#59c7fa"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <com.github.mikephil.charting.charts.LineChart
        android:id="@+id/temp_chart"
        android:layout_weight="1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

    <TextView
        android:id="@+id/hum_textView"
        android:textColor="@android:color/white"
        android:textSize="40sp"
        android:textStyle="bold"
        android:padding="10sp"
        android:background="#fa6868"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <com.github.mikephil.charting.charts.LineChart
        android:id="@+id/hum_chart"
        android:layout_weight="1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</LinearLayout>
```

Listing 2. Uzyskanie dostępu do magistrali I2C i modułu HTU21private I2cDevice htu21

```
PeripheralManagerService service = new PeripheralManagerService();
/* open HTU21 device */
try {
    htu21 = service.openI2cDevice("I2C1", 0x40);
} catch (IOException e) {
    Log.w(TAG, "Unable to access HTU21 device", e);
}
```

Tabela 1. Wybrane metody klasy I2cDevice [4]

Funkcja	Zadanie
abstract void close ()	Zamknij urządzenie i zwolnij zajmowane zasoby.
abstract void read (byte[] buffer, int length)	Odczytaj określoną liczbę bajtów danych (length) z urządzenia I²C do tablicy (buffer).
abstract void readRegBuffer (int reg, byte[] buffer, int length)	Odczytaj określoną liczbę bajtów danych do tablicy buffer, z rejestru o adresie reg.
abstract byte readRegByte (int reg)	Odczytaj jeden bajt danych z rejestru o adresie reg.
abstract short readRegWord (int reg)	Odczytaj dwa bajty danych z rejestru o adresie reg.
abstract void write(byte[] buffer, int length)	Zapisz określoną liczbę bajtów danych (length) z tablicy (buffer) do urządzenia I²C.
abstract void writeRegBuffer (int reg, byte[] buffer, int length)	Zapisz określoną liczbę bajtów danych (length) z tablicy (buffer) pod określony rejestr (reg) urządzenia I²C.
abstract void writeRegByte (int reg, byte data)	Zapisz jeden bajt danych do urządzenia I²C pod rejestr o adresie reg.
abstract void writeRegWord (int reg, short data)	Zapisz dwa bajty danych do urządzenia I²C pod rejestr o adresie reg.

Tabela 2. Lista komend sterujących pracą modułu pomiarowego HTU21 [5]

Komenda	Kod komendy	Opis
Rozpoczęcie pomiaru temperatury w blokującym trybie "Hold Master"	0xE3	Komenda rozpoczyna pomiar temperatury w trybie blokującym – linia SCK jest ściągana przez układ HTU21 do czasu realizacji pomiaru – układ Master nie może w tym czasie przeprowadzić innych działań na magistrali. Po zakończeniu pomiaru, układ HTU21 przesyła trzy bajty danych: dwa pierwsze bajty (w kolejności odpowiednio MSB i LSB) stanowią wyniki pomiaru, natomiast trzeci bajt stanowi sumę kontrolną CRC.
Rozpoczęcie pomiaru wilgotności w blokującym trybie "Hold Master"	0xE5	Komenda rozpoczyna pomiar wilgotności w trybie blokującym – linia SCK jest ściągana przez układ HTU21 do czasu realizacji pomiaru – układ Master nie może w tym czasie przeprowadzić innych działań na magistrali. Po zakończeniu pomiaru, układ HTU21 przesyła trzy bajty danych: dwa pierwsze bajty (w kolejności odpowiednio MSB i LSB) stanowią wyniki pomiaru, natomiast trzeci bajt stanowi sumę kontrolną CRC.
Rozpoczęcie pomiaru temperatury w trybie nieblokującym	0xF3	Komenda 0xF3 rozpoczyna pomiar temperatury w trybie bez blokowania linii SCK. Wówczas to, zadaniem układu Master jest odpytanie modułu HTU21 o status zakończenia pomiaru, co jest sygnalizowane poprzez odpowiednie ustawienie bitu ACK lub NACK, w odpowiedzi na przesłanie adresu urządzenia z bitem kierunku transmisji ustawionym jako '1' – Read.
Rozpoczęcie pomiaru wilgotności w trybie nieblokującym	0xF5	Komenda 0xF5 rozpoczyna pomiar wilgotności w trybie bez blokowania linii SCK. Wówczas to, zadaniem układu Master jest odpytanie modułu HTU21 o status zakończenia pomiaru, co jest sygnalizowane poprzez odpowiednie ustawienie bitu ACK lub NACK, w odpowiedzi na przesłanie adresu urządzenia z bitem kierunku transmisji ustawionym jako '1' – Read.
Zapis rejestru konfiguracji urządzenia	0xE6	Komenda 0xE6 umożliwia zapis danych do rejestru konfiguracyjnego modułu, a tym samym zmianę rozdzielczości pomiarowej (domyślnie: 12-bitów dla pomiaru wilgotności oraz 14-bitów dla pomiaru temperatury) a także sprawdzenie stanu napięcia na linii VDD czy włączenie opcji „podgrzania", umożliwiającej diagnozę pracy czujnika.
Odczyt rejestru konfiguracji urządzenia	0xE7	Komenda umożliwiająca odczyt danych z rejestru konfiguracyjnego.
Programowy RESET urządzenia	0xFE	Komenda 0xFE pozwala na wykonanie programowego resetu modułu HTU21.

- S_{RH} – 16-bitowa wartość odczytana z modułu po przesłaniu komendy 0xE5 lub 0xF5.

Implementacja funkcji realizującej wyzwolenie pomiarów oraz przeliczenie mierzonych wartości, została przedstawiona na **listingu 6**.

Jak można zauważyć, mimo braku dedykowanego sterownika, podstawowa obsługa układu *HTU21*, to tylko kilkanaście linii kodu. Dzięki funkcjom udostępnianym przez *Things Support Library*, programista nie musi zapewniać pełnej obsługi kontrolera interfejsu I²C (poprzez bezpośrednią pracę na rejestrach, kontrolę bitów *ACK/NACK*, itd.). Jeszcze lepiej sytuacja prezentuje się w momencie, gdy w systemie mamy dostępny gotowy sterownik urządzenia. Dodajmy więc zatem do naszego projektu prostą obsługę wyświetlacza z kontrolerem *SSD1306*, z wykorzystaniem sterownika udostępnionego przez bibliotekę *Peripheral Driver Library*.

Dzięki wykorzystaniu gotowego sterownika, programista nie musi znać żadnych aspektów niskopoziomowej obsługi wyświetlacza (adresu na magistrali, definicji rejestrów, itp.) – cała jego programowa obsługa została zawarta w kilku dobrze zdefiniowanych metodach obiektu *Ssd1306* [6] – **tabela 3**.

Ponieważ nasz projekt posiada już zdefiniowany graficzny interfejs użytkownika (wyświetlany z pomocą ekranu dołączanego do portu HDMI), niech obsługa kontrolera *SSD1306* ma charakter wyłącznie dydaktyczny – zadaniem miniaturowego wyświetlacza *OLED* będzie wyłącznie zaprezentowanie prostego wzoru graficznego (korzystając z informacji zawartych w ramce poniżej, można nadać bardziej praktyczny charakter zadań przypisanych do wyświetlacza *SSD1306*, poprzez wyświetlenie „logo producenta" lub napisów informujących o błędach w odczycie czujnika *HTU21*). Kod implementujący wyświetlenie prostego wzoru szachownicy został przedstawiony na **listingu 7**.

Ostatnim elementem do pełnego skompletowania projektu jest konfiguracja wykresów – generowanych z wykorzystaniem biblioteki *MPAndroidChart* – oraz programowa obsługa interfejsu

Listing 3. Generyczna funkcja dla pomiarów temperatury i wilgotności w trybie „Hold Master"

```
private float htu21_readData (I2cDevice htu21, int htu21_cmd) throws Exception {
    byte[] cmd = {(byte) htu21_cmd};
    byte[] output = new byte[3];
    int data_with_crc, polynomial;
    /* trigger measurement - ‚hold Master' mode */
    htu21.write(cmd, 1);
    htu21.read(output, 3);
    /* check CRC */
    data_with_crc = output[0] << 16 | (output[1] & 0xFF) << 8 | (output[2] & 0xFF);
    polynomial = 0x98800000; /* CRC Polynomial: x^8 + x^5 + x^4 + 1 */
    for (int cnt = 0; cnt < 24; cnt++) {
        if ((data_with_crc & 0x80000000) != 0) data_with_crc ^= polynomial;
        data_with_crc <<= 1;
    }
    if (data_with_crc != 0) throw new Exception („CRC Error");
    return (output[0] << 8 | (output[1] & 0xFF));
}
```

Listing 4. Inicjalizacja czujnika HTU21 w metodzie onCreate()

```
if (htu21 != null) {
    if (htu21_softReset(htu21)) {
        htu21_handler.postDelayed(htu21_runnable, 1000);
    } else {
        /* ERROR ! */
        / .. /
    }
}
```

Listing 5. Kod funkcji htu21_softReset()

```
private boolean htu21_softReset (I2cDevice htu21) {
    byte[] cmd = {(byte) 0xFE};
    /* trigger measurement - ‚hold Master' mode */
    try {
        htu21.write(cmd, 1);
    } catch (IOException e) {
        return false;
    }
    return true;
}
```

Listing 6. Kod funkcji htu21_softReset()

```
private Runnable htu21_runnable = new Runnable() {
    @Override
    public void run() {
        float temp;
        float hum;
        /* read data from HTU21 */
        try {
            temp = htu21_readData(htu21, (byte) 0xE3);
            hum = htu21_readData(htu21, (byte) 0xE5);
            temp *= 175.72;
            temp /= 65536;
            temp -= 46.85;
            hum *= 125;
            hum /= 65536;
            hum -= 6;
                /* AKTUALIZACJA GUI */
                    / ... /
        } catch (Exception e) {
            e.printStackTrace();
        }
        /* schedule another event after 1 sec. delay */
        htu21_handler.postDelayed(htu21_runnable, 1000);
    }
};
```

Tabela 3. Wybrane metody klasy Ssd1306 [6]

Funkcja	Zadanie
int getLcdWidth ()	Funkcja zwraca szerokość wyświetlacza wyrażoną w pikselach.
int getLcdHeight ()	Funkcja zwraca wysokość wyświetlacza wyrażoną w pikselach.
void clearPixels ()	Funkcja czyści zawartość lokalnego bufora danych, która jest przesyłana do wyświetlacza po wywołaniu metody show().
void setPixel (int x, int y, boolean on)	Funkcja zapala (gdy argument on przyjmuje wartość TRUE) lub gasi (gdy aragument on przyjmuje wartość FALSE) wybrany piksel o współrzędnych x,y. Zmiana dokonywana jest w lokalnym buforze danych – do aktualizacji wyświetlacza niezbędne jest wywołanie funkcji show().
void setDisplayOn (boolean on)	Funkcja włączająca/wyłączająca wyświetlacz.
void show ()	Funkcja przesyłająca aktualną zawartość lokalnego bufora danych do kontrolera SSD1306.
void startScroll (int startY, int finishY, ScrollMode scrollMode)	Sprzętowa realizacja efektu "przewijania" zawartości wyświetlacza od wiersza startY do wiersza finishY. Efekt przewijania może zostać zrealizowany w jednym z poniższych trybów: COMMAND_RIGHT_HORIZONTAL_SCROLL, COMMAND_LEFT_HORIZONTAL_SCROLL, COMMAND_VERTICAL_AND_LEFT_HORIZONTAL_SCROLL lub COMMAND_VERTICAL_AND_RIGHT_HORIZONTAL_SCROLL.
void stopScroll ()	Zatrzymanie efektu „przewijania" ekranu wyświetlacza.

graficznego. W metodzie *onCreate()*, wywołanie *setContentView()* pobiera dane z pliku XML opisującego układ (w omawianym przypadku jest to plik *layout/activity_main.xml*), a następnie tworzy obiekty dla odpowiednich komponentów. Za pomocą szeregu wywołań *findViewById()*, pobieramy referencję do tych obiektów, co pozwoli następnie na zmianę wybranych ich wartości np. poprzez metodę *setText()* – **listing 8**. Po serii wywołań *findViewById()*, program wykonuje utworzoną na potrzeby projektu funkcję *setupChart()*, która konfiguruje pola wykresów dla pomiarów temperatury i wilgotności – **listing 9**.

Biblioteka *MPAndroidChart*, nie jest integralną częścią systemu *Android Things*, tak więc niniejszy artykuł nie będzie zawierał opisu jej konfiguracji i obsługi. Szczegółowy opis interfejsu programowego (*API*) biblioteki został umieszczony na stronie projektu pod adresem ***https://goo.gl/UAoif8***. Do głównych zadań funkcji *setupChart()* należy konfiguracja osi wykresu (ustawienie wartości maksymalnej i minimalnej na osi Y, wielkości czcionki dla wartości opisujących oś, itp.), a także koloru tła i ustawień siatki. Wykresy inicjalizowane są pustym zbiorem wartości (poprzez metodę *setData()*), który wraz z realizacją kolejnych pomiarów, jest uzupełniany poprzez wywołanie *addEntry()*, jak na **listingu 10**.

Finalny efekt projektu *AndroidThings-I2C* zaprezentowano na poniższym krótkim filmie ***https://goo.gl/HpvKCB***. Uporządkowany i skompletowany (również o nadpisanie metody *onDestroy()*) kod programu został przedstawiony na **listingu 11**.

Łukasz Skalski

Linki zewnętrzne:

https://goo.gl/ViJh6x
https://goo.gl/ZtuUp7
https://goo.gl/bMjjnD
https://goo.gl/EvtVhe
https://goo.gl/AB3xCN
https://goo.gl/732BYY

#Listing 7. Wyświetlenie prostego wzoru graficznego z wykorzystaniem sterownika SSD1306

```
try {
    Ssd1306 ssd1306 = new Ssd1306("I2C1");
    for (int i = 0; i < ssd1306.getLcdWidth(); i++) {
        for (int j = 0; j < ssd1306.getLcdHeight(); j++) {
            ssd1306.setPixel(i, j, (i % 2) == (j % 2));
        }
    }
    ssd1306.show();
} catch (IOException e) {
    Log.e(TAG, "Error while opening sdd1306 display", e);
}
```

Listing 8. Inicjalizacja komponentów graficznego interfejsu użytkownika

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    temp_chart = findViewById(R.id.temp_chart);
    hum_chart = findViewById(R.id.hum_chart);
    temp_textView = findViewById(R.id.temp_textView);
    hum_textView = findViewById(R.id.hum_textView);
    /* setup charts */
    setupChart (temp_chart, Color.parseColor("#59c7fa"), Color.parseColor("#9cdefc"), 15, 35);
    setupChart (hum_chart, Color.parseColor("#fa6868"), Color.parseColor("#fc9c9c"), 45, 65);
    / ... /
}
```

Biblioteka *Peripheral Driver Library* wraz ze sterownikiem dla kontrolera *SSD1306* udostępnia również klasę pomocniczą *BitmapHelper*, dzięki której użytkownik ma możliwość wyświetlenia na ekranie dowolnej grafiki zapisanej w postaci bitmapy:

```
ssd1306 = new Ssd1306("I2C1");
Bitmap bmp = BitmapFactory.decodeResource(getResources(), R.drawable.logo);
BitmapHelper.setBmpData(ssd1306, 0, 0, bmp, false);
ssd1306.show();
```

gdzie *R.drawable.logo*, odnosi się do monochromatycznego pliku graficznego *logo.png* o wymiarach zgodnych z rozmiarami wyświetlacza, i umieszczonego w katalogu *app/src/main/res/drawable*.
Ponieważ biblioteka wyświetlacza nie udostępnia gotowych funkcji związanych z wyświetlaniem napisów, użytkownik może wykorzystać metodę *setBmpData()* oraz funkcje tworzące proste bitmapy z napisami, jako „niecodzienny" sposób implementacji wyświetlania napisów:

```
String text = "NAPIS!";
Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);
paint.setTextSize(30f);
paint.setColor(Color.WHITE);
paint.setTextAlign(Paint.Align.LEFT);
Bitmap textAsBitmap = Bitmap.createBitmap(128, 64, ARGB_8888);
Canvas canvas = new Canvas(textAsBitmap);
canvas.drawText(text, 0, 0.5f * 64, paint);
BitmapHelper.setBmpData(ssd1306, 0, 0, textAsBitmap, true);
```

Listing 11. Kompletny kod klasy MainActivity

```
package com.skalski.lukasz.androidthings_i2c;
import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Color;
import android.os.Bundle;
import android.os.Handler;
import android.util.Log;
import android.widget.TextView;
import java.io.IOException;
import java.util.Locale;
import com.google.android.things.contrib.driver.ssd1306.BitmapHelper;
import com.google.android.things.contrib.driver.ssd1306.Ssd1306;
import com.google.android.things.pio.I2cDevice;
import com.google.android.things.pio.PeripheralManagerService;
import com.github.mikephil.charting.charts.LineChart;
import com.github.mikephil.charting.components.Legend;
import com.github.mikephil.charting.data.Entry;
import com.github.mikephil.charting.data.LineData;
import com.github.mikephil.charting.data.LineDataSet;
import com.github.mikephil.charting.interfaces.datasets.ILineDataSet;

public class MainActivity extends Activity {
    private static final String TAG = „MainActivity”;
    private LineChart temp_chart;
    private LineChart hum_chart;
    private TextView temp_textView;
    private TextView hum_textView;
    private Ssd1306 ssd1306;
    private I2cDevice htu21;
    private Handler htu21_handler = new Handler();
    private static int HTU21_ADDRESS = 0x40;
    private static int HTU21_TEMP_HOLD_MODE = 0xE3;
    private static int HTU21_HUM_HOLD_MODE = 0xE5;
    private static int HTU21_SOFT_RESET = 0xFE;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        temp_chart = findViewById(R.id.temp_chart);
        hum_chart = findViewById(R.id.hum_chart);
        temp_textView = findViewById(R.id.temp_textView);
        hum_textView = findViewById(R.id.hum_textView);
        /* setup charts */
        setupChart(temp_chart, Color.parseColor(„#59c7fa”), Color.parseColor(„#9cdefc”), 15, 35);
        setupChart(hum_chart, Color.parseColor(„#fa6868”), Color.parseColor(„#fc9c9c”), 45, 65);
        /* connect to ssd1306 display */
        PeripheralManagerService service = new PeripheralManagerService();
        try {
            ssd1306 = new Ssd1306(„I2C1”);
            Bitmap bmp = BitmapFactory.decodeResource(getResources(), R.drawable.logo);
            BitmapHelper.setBmpData(ssd1306, 0, 0, bmp, false);
            ssd1306.show();
        } catch (IOException e) {
            Log.e(TAG, „Error while opening sdd1306 display”, e);
        }
        /* open HTU21 device */
        try {
            htu21 = service.openI2cDevice(„I2C1”, HTU21_ADDRESS);
        } catch (IOException e) {
            Log.w(TAG, „Unable to access HTU21 device”, e);
        }
        /* send RESET to HTU21 */
        if (htu21 != null) {
            if (htu21_softReset(htu21)) {
                htu21_handler.postDelayed(htu21_runnable, 1000);
            } else {
                temp_textView.setText(getString(R.string.htu21_error));
                hum_textView.setText(getString(R.string.htu21_error));
            }
        }
    }
    /*
     * htu21_runnable
     */
    private Runnable htu21_runnable = new Runnable() {
        @Override
        public void run() {
            float temp;
            float hum;
            /* read data from HTU21 */
            try {
                temp = htu21_readData(htu21, HTU21_TEMP_HOLD_MODE);
                hum = htu21_readData(htu21, HTU21_HUM_HOLD_MODE);
                temp *= 175.72;
                temp /= 65536;
                temp -= 46.85;
                hum *= 125;
                hum /= 65536;
                hum -= 6;
                /* update charts */
                addEntry(temp_chart, temp);
                addEntry(hum_chart, hum);
                /* update textViews */
                temp_textView.setText(„Temperature: „ + String.format(Locale.US, „%.1f”, temp) + „ \u2103”);
                hum_textView.setText(„Humidity: „ + (int)hum + „%”);
            } catch (Exception e) {
                e.printStackTrace();
            }
            /* schedule another event after 1 sec. delay */
            htu21_handler.postDelayed(htu21_runnable, 1000);
        }
    };
    /*
     * htu21_readData()
     */
    private float htu21_readData (I2cDevice htu21, int htu21_cmd) throws Exception {
        byte[] cmd = {(byte) htu21_cmd};
        byte[] output = new byte[3];
        int data_with_crc, polynomial;
        /* trigger measurement - ‚hold Master’ mode */
        htu21.write(cmd, 1);
        htu21.read(output, 3);
        /* check CRC */
        data_with_crc = output[0] << 16 | (output[1] & 0xFF) << 8 | (output[2] & 0xFF);
```

Listing 9. Konfiguracja wykresów LineChart w funkcji setupChart()

```
private void setupChart (LineChart chart,
    int bg_color, int grid_color,
    int axis_min, int axis_max) {
    /* disable all ‚interactions’ with the chart */
    chart.setTouchEnabled(false);
    chart.setDragEnabled(false);
    chart.setScaleEnabled(false);
    chart.setPinchZoom(false);
    /* configure axis */
    chart.getAxisRight().setEnabled(false);
    chart.getXAxis().setEnabled(false);
    /* configure AxisLeft */
    chart.getAxisLeft().setEnabled(true);
    chart.getAxisLeft().setTextColor(Color.WHITE);
    chart.getAxisLeft().setTextSize(15f);
    chart.getAxisLeft().setGridColor(grid_color);
    chart.getAxisLeft().setGridLineWidth(1f);
    chart.getAxisLeft().setGranularity(0.5f);
    chart.getAxisLeft().setAxisMinimum(axis_min);
    chart.getAxisLeft().setAxisMaximum(axis_max);
    chart.getDescription().setEnabled(false);
    chart.setDrawGridBackground(false);
    chart.setBackgroundColor(bg_color);
    chart.setViewPortOffsets(70, 30, 70, 30);
    chart.setData(new LineData());
    /* don’t show legend */
    Legend l = chart.getLegend();
    l.setEnabled(false);
}
```

Listing 11. cd.

```
        polynomial = 0x98800000; /* CRC Polynomial: x^8 + x^5 + x^4 + 1 */
        for (int cnt = 0; cnt < 24; cnt++) {
            if ((data_with_crc & 0x80000000) != 0)
                data_with_crc ^= polynomial;
            data_with_crc <<= 1;
        }
        if (data_with_crc != 0)
            throw new Exception („CRC Error”);
        return (output[0] << 8 | (output[1] & 0xFF));
    }
    /*
     * htu21_softReset()
     */
    private boolean htu21_softReset (I2cDevice htu21) {
        byte[] cmd = {(byte) HTU21_SOFT_RESET};
        /* trigger measurement - ‚hold Master’ mode */
        try {
            htu21.write(cmd, 1);
        } catch (IOException e) {
            return false;
        }
        return true;
    }
    /*
     * onDestroy()
     */
    @Override
    protected void onDestroy() {
        super.onDestroy();

        /* remove handler events on close */
        htu21_handler.removeCallbacks(htu21_runnable);
        /* close I2C device */
        if (htu21 != null) {
            try {
                htu21.close();
                htu21 = null;
            } catch (IOException e) {
                Log.w(TAG, „Unable to close I2C device”, e);
            }
        }
        /* close SSD1306 device */
        if (ssd1306 != null) {
            try {
                ssd1306.close();
                ssd1306 = null;
            } catch (IOException e) {
                Log.w(TAG, „Unable to close SSD1306 device”, e);
            }
        }
    }
    /*
     * setupChart()
     */
    private void setupChart (LineChart chart, int bg_color, int grid_color, int axis_min, int axis_max) {
        /* disable all ‚interactions’ with the chart */
        chart.setTouchEnabled(false);
        chart.setDragEnabled(false);
        chart.setScaleEnabled(false);
        chart.setPinchZoom(false);
        /* configure axis */
        chart.getAxisRight().setEnabled(false);
        chart.getXAxis().setEnabled(false);
        /* configure AxisLeft */
        chart.getAxisLeft().setEnabled(true);
        chart.getAxisLeft().setTextColor(Color.WHITE);
        chart.getAxisLeft().setTextSize(15f);
        chart.getAxisLeft().setGridColor(grid_color);
        chart.getAxisLeft().setGridLineWidth(1f);
        chart.getAxisLeft().setGranularity(0.5f);
        chart.getAxisLeft().setAxisMinimum(axis_min);
        chart.getAxisLeft().setAxisMaximum(axis_max);
        chart.getDescription().setEnabled(false);
        chart.setDrawGridBackground(false);
        chart.setBackgroundColor(bg_color);
        chart.setViewPortOffsets(70, 30, 70, 30);
        chart.setData(new LineData());
        /* don’t show legend */
        Legend l = chart.getLegend();
        l.setEnabled(false);
    }
    /*
     * createSet()
     */
    private LineDataSet createSet() {
        LineDataSet dataSet = new LineDataSet(null, „Test”);
        dataSet.setLineWidth(2f);
        dataSet.setCircleRadius(5f);
        dataSet.setCircleHoleRadius(5f);
        dataSet.setColor(Color.WHITE);
        dataSet.setCircleColor(Color.WHITE);
        dataSet.setDrawValues(false);
        return dataSet;
    }
    /*
     * addEntry()
     */
    private void addEntry (LineChart mChart, float value) {
        LineData data = mChart.getData();
        if (data != null) {
            ILineDataSet set = data.getDataSetByIndex(0);
            if (set == null) {
                set = createSet();
                data.addDataSet(set);
            }
            data.addEntry(new Entry(set.getEntryCount(), value), 0);
            data.notifyDataChanged();
            /* let the chart know it’s data has changed */
            mChart.notifyDataSetChanged();
            /* limit the number of visible entries */
            mChart.setVisibleXRangeMaximum(60);
            /* move to the latest entry */
            mChart.moveViewToX(data.getEntryCount());
        }
    }
}
```

Listing 10. Funkcja addEntry() dodająca nowe pomiary do wykresów temperatury i wilgotności

```
private void addEntry (LineChart mChart, float value) {
    LineData data = mChart.getData();
    if (data != null) {
        ILineDataSet set = data.getDataSetByIndex(0);
        if (set == null) {
            set = createSet();
            data.addDataSet(set);
        }
        data.addEntry(new Entry(set.getEntryCount(), value), 0);
        data.notifyDataChanged();
        /* let the chart know it’s data has changed */
        mChart.notifyDataSetChanged();
        /* limit the number of visible entries */
        mChart.setVisibleXRangeMaximum(60);
        /* move to the latest entry */
        mChart.moveViewToX(data.getEntryCount());
    }
}
```

Poprzednie części kursu i dodatkowe materiały dostępne są na FTP: ftp://ep.com.pl, user: 95777, pass: 53wtjyf6