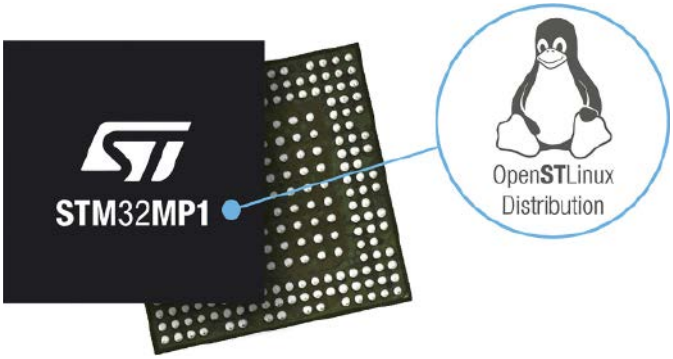
OpenSTLinux

dla procesorów z rodziny STM32MP1 (6)

FTP WSZYSTKIE POPRZEDNIE ODCINKI KURSU SĄ DOSTĘPNE NA STRONIE WWW.MEDIA.AVT.PL

W ostatnim odcinku kursu poświęconego procesorom STM32MP1 zajmiemy się opracowaniem aplikacji z graficznym interfejsem użytkownika. Przygotujemy przycisk i suwak do sterowania diodami oraz wygenerujemy wykres na bazie liczb pseudolosowych. Wykorzystamy do tego biblioteki Qt, które zbudujemy razem z obrazem systemu OpenSTLinux oraz SDK.



Kompilacja bibliotek Qt i SDK

Do implementacji aplikacji będziemy potrzebowali odpowiednio zmodyfikowanego SDK oraz obrazu systemu, na którym będziemy mogli ją uruchomić. Bazowe biblioteki Qt są domyślnie dodane w warstwie meta-somlabs, w pliku *recipes-st/images/st-image-weston.bbappend*, jednak dodatkowo będziemy potrzebować obsługi wykresów. W Qt wykresy zostały zaimplementowane w module *Qt Charts*, który możemy dodać do naszego obrazu dopisując `qtcharts` do listy instalowanych pakietów w pliku *st-image-weston.bbappend*, tak jak na **listingu 1**.

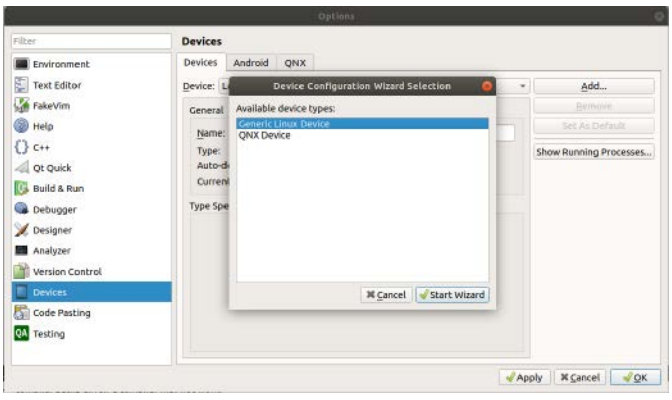
Musimy teraz przebudować zarówno system, jak i SDK, tak jak to już omawialiśmy w poprzednich odcinkach kursu. W dalszej części tekstu przyjmujemy założenie, że SDK zostało zainstalowane w systemie bazowym w domyślnym katalogu */opt/st/stm32mp157a-visionsom-rgb-sd-mx/3.1-snapshot*.

Qt Creator IDE

Do wykonania projektu użyjemy środowiska Qt Creator, które będziemy musieli najpierw odpowiednio skonfigurować, aby do kompilacji wykorzystane zostały biblioteki i narzędzia z SDK. Zaprezentowany opis dotyczy środowiska Qt Creator w wersji 4.5.2 zainstalowanego w systemie Ubuntu 18.04 za pomocą polecenia `sudo apt-get install qtcreator`.

Listing 1. Lista pakietów w pliku *st-image-weston.bbappend*, z dodanym wpisem *qtcharts*

```
inherit populate_sdk_qt5

IMAGE_INSTALL += " \
    qtbase-dev \
    qtbase-mkspecs \
    qtbase-tools \
    qtdeclarative-qmlplugins \
    qtquickcontrols2-qmlplugins \
    qtwayland \
    gstreamer1.0 \
    gstreamer1.0-plugins-good \
    qtcharts \
"
```



Rysunek 1. Dodawanie nowego urządzenia

Aby uruchomić IDE musimy otworzyć najpierw terminal i wywołać w nim polecenie ustawiające zmienne środowiskowe dla SDK. Następnie w tym samym terminalu uruchamiamy Qt Creatora:

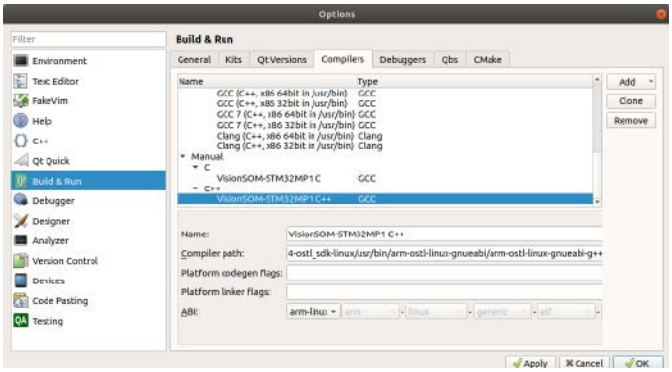
```
. /opt/st/stm32mp157a-visionsom-rgb-sd-mx/3.1-snapshot/environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi
qtcreator
```

Konfigurację Qt Creatora zaczynamy od dodania urządzenia wybierając menu Tools → Options i przechodząc do Devices. W oknie dialogowym klikamy Add i wybieramy Generic Linux Device (**rysunek 1**). Po uruchomieniu kreatora przyciskiem Start Wizard musimy uzupełnić dane urządzenia – nazwę oraz adres IP, dzięki któremu Qt Creator będzie mógł się z nim połączyć (**rysunek 2**). Do tego celu możemy wybrać dowolny dostępny interfejs, łącznie z usb0, który omawialiśmy w poprzednim odcinku kursu. Po zakończeniu konfiguracji urządzenia jest automatycznie przeprowadzany test komunikacji, dzięki któremu dowiemy się, czy poprawnie wprowadziliśmy dane (**rysunek 3**).

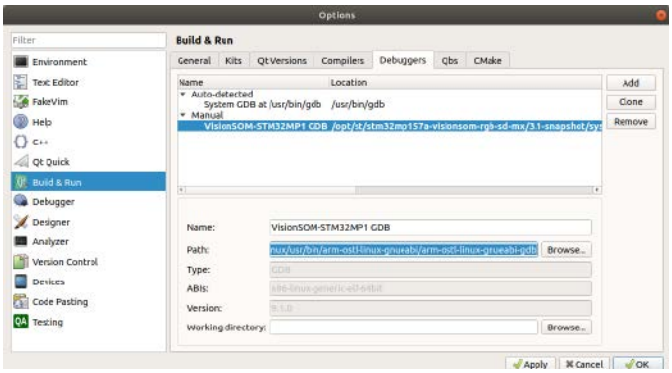
Następnym krokiem jest dodanie ścieżek kompilatorów C/C++ w opcji Build & Run → Compilers. Należy je dodać ręcznie wybierając przycisk Add → GCC → C i Add → GCC → C++. Każdemu z nich powinniśmy nadać nazwę i podać ścieżkę do zainstalowanego SDK: */opt/st/stm32mp157a-visionsom-rgb-sd-mx/3.1-snapshot/*

Rysunek 2. Nazwa i adres podłączonego urządzenia

Rysunek 3. Test komunikacji zakończony sukcesem



Rysunek 4. Dodanie kompilatorów C i C++ z SDK



Rysunek 5. Dodanie GDB z SDK

sysroots/x86_64-ostl_sdk-linux/usr/bin/arm-ostl-linux-gnueabi/arm-ostl-linux-gnueabi-gcc

oraz

/opt/st/stm32mp157a-visionsom-rgb-sd-mx/3.1-snapshot/sysroots/x86_64-ostl_sdk-linux/usr/bin/arm-ostl-linux-gnueabi/arm-ostl-linux-gnueabi-g++.

W polu ABI należy wybrać *arm-linux-generic-elf-32bit*. Dodane kompilatory pokazane są na **rysunku 4**. Opcjonalnie możemy dodać także GDB w zakładce Debuggers podając ścieżkę: */opt/st/stm32mp157a-visionsom-rgb-sd-mx/3.1-snapshot/sysroots/x86_64-ostl_sdk-linux/usr/bin/arm-ostl-linux-gnueabi/arm-ostl-linux-gnueabi-gdb*, tak jak to pokazano na **rysunku 5**.

Biblioteki Qt z SDK powinny zostać wykryte automatycznie, co możemy zweryfikować w zakładce Qt Versions. Wersją właściwą dla użytego SDK jest 5.14.2. Jeżeli Qt nie zostanie wykryte automatycznie możemy ręcznie dodać ścieżkę do pliku qmake: */opt/st/stm32mp157a-visionsom-rgb-sd-mx/3.1-snapshot/sysroots/x86_64-ostl_sdk-linux/usr/bin/qmake*.

Ostatnim krokiem konfiguracji jest utworzenie Kitu w zakładce Kits. Po kliknięciu Add uzupełniamy nazwę, typ urządzenia (Generic Linux Device), nazwę urządzenia, które dodaliśmy wcześniej, ścieżkę do sysroot (*/opt/st/stm32mp157a-visionsom-rgb-sd-mx/3.1-snapshot/sysroots/cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi*), kompilatory C/C++, debugger i wersję Qt. Wypełniony formularz został pokazany na **rysunku 6**.

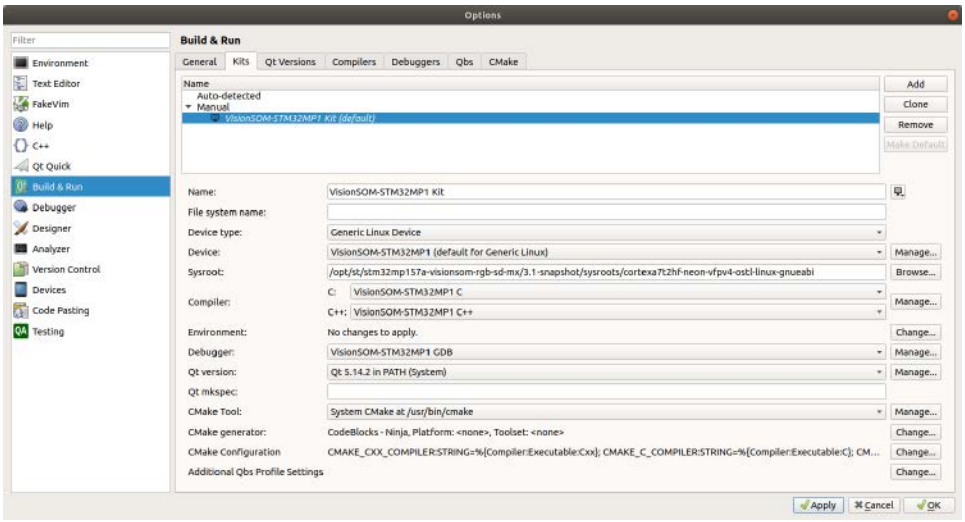
Tworzenie nowego projektu

Mając skonfigurowane środowisko możemy rozpocząć tworzenie przykładowej aplikacji na bazie gotowego przykładu z biblioteki.

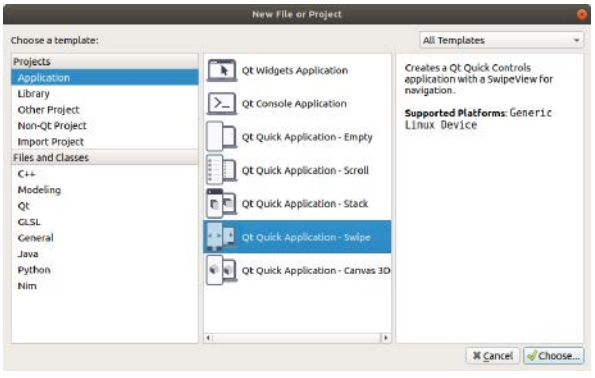
Listing 2. Opis pierwszej strony aplikacji

```
import QtQuick 2.9
import QtQuick.Controls 2.2

Page {
    property alias button: button
    property alias slider: slider

    header: Label {
        text: qsTr("Page 1")
        font.pixelSize: Qt.application.font.pixelSize * 2
        padding: 10
    }

    Button {
        id: button
        y: 110
        width: 120
        height: 60
        text: qsTr("LED")
        anchors.left: parent.left
        anchors.leftMargin: 100
        anchors.verticalCenter: parent.verticalCenter
        checkable: true
    }

    Slider {
        id: slider
        x: 260
        y: 110
        width: 300
        anchors.right: parent.right
        anchors.rightMargin: 100
        anchors.verticalCenterOffset: 0
        anchors.verticalCenter: parent.verticalCenter
        from: 0
        to: 100
        stepSize: 10
        value: 0
    }
}
```

Z menu File wybieramy New File or Project, a w wyświetlonym oknie dialogowym opcję Qt Quick Application – Swipe (**rysunek 7**). Następnie nadajemy projektowi nazwę – w naszym przykładzie jest to SwipeDemo, oraz lokalizację na dysku. Kolejne opcje możemy zostawić bez zmiany, zwracając uwagę na to, żeby do projektu został przydzielony Kit, który konfigurowaliśmy w poprzednim punkcie. Po zakończeniu



Rysunek 6. Konfiguracja Kitu dla VisionSOM-STM32MP1



Rysunek 7. Tworzenie nowego projektu z szablonu *Swipe*

Listing 3. Opis drugiej strony aplikacji

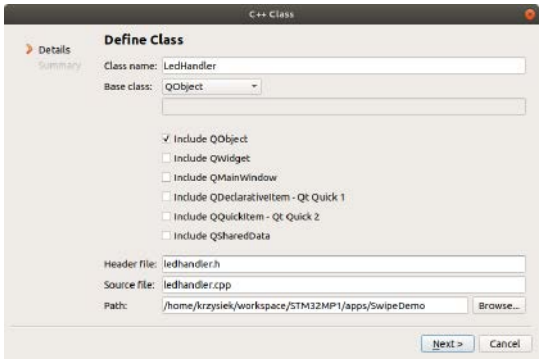
```
import QtQuick 2.9
import QtQuick.Controls 2.2
import QtCharts 2.0

ChartView {
    antialiasing: true
    animationOptions: ChartView.NoAnimation

    property int numberOfPoints: 0
    property int maxNumberOfPoints: 100
    property alias dataSeries: dataSeries

    LineSeries {
        id: dataSeries
        name: "Values"
        useOpenGL: true

        property alias xAxis: xAxis

        axisX: ValueAxis {
            id: xAxis
            min: -1 * maxNumberOfPoints
            max: 0
            visible: false
        }
        axisY: ValueAxis {
            min: 0
            max: 100
        }
    }
}
```

działania kreatora zostaje utworzony nowy projekt zawierający okno główne z dwiema stronami które możemy przesuwać za pomocą myszy, lub panelu dotykowego. W projekcie znajdziemy kilka plików, które będziemy mogli zmodyfikować:

- *main.cpp* – kod C++ aplikacji,
- *main.qml* – opis wyglądu głównego okna zapisany w języku QML,
- *Page1Form.ui.qml* – opis wyglądu pierwszej strony aplikacji,
- *Page2Form.ui.qml* – opis wyglądu drugiej strony aplikacji,
- *qtquickcontrols2.conf* – opis stylu komponentów graficznych,
- *SwipeDemo.pro* – konfiguracja projektu.

Listing 4. Opis głównego okna aplikacji

```
import QtQuick 2.9
import QtQuick.Controls 2.2

ApplicationWindow {
    visible: true
    visibility: "FullScreen"
    title: qsTr("Tabs")

    SwipeView {
        id: swipeView
        anchors.fill: parent
        currentIndex: tabBar.currentIndex

        Page1Form {
            button.onCheckedChanged: ledHandler.setEnabled(button.checked)
            slider.onValueChanged: ledHandler.setBrightness(slider.value)
        }

        Page2Form {
            id: chartPage
            Connections {
                target: pointGenerator
                onDoAddSample: {
                    chartPage.addPoint(value)
                }
            }

            function addPoint(value) {
                dataSeries.append(numberOfPoints, value)

                if (dataSeries.count > maxNumberOfPoints) {
                    dataSeries.remove(0)
                }

                dataSeries.xAxis.min += 1
                dataSeries.xAxis.max += 1
                numberOfPoints += 1
            }
        }
    }

    footer: TabBar {
        id: tabBar
        currentIndex: swipeView.currentIndex

        TabButton {
            text: qsTr("Page 1")
        }
        TabButton {
            text: qsTr("Page 2")
        }
    }
}
```



Rysunek 8. Dodanie klasy *LedHandler* do projektu

Interfejs graficzny możemy definiować na dwa sposoby. Pierwszy z nich to edytor graficzny (przycisk Design po lewej stronie okna głównego IDE) dla plików *ui.qml*, a drugi to opis w języku QML. Obie metody się uzupełniają, ponieważ możemy ręcznie modyfikować zmiany wprowadzone przez edytor graficzny i na odwrót. W naszym przykładzie wszystkie zmiany wprowadzimy ręcznie. Na początku modyfikujemy plik *Page1Form.ui.qml*, tak jak pokazano na **listingu 2**. W kodzie QML dodajemy przycisk oraz suwak, a także definiujemy ich rozmiary i rozmieszczenie na stronie. Na początku struktury Page dodajemy także aliasy dla utworzonych komponentów, tak aby można było się do nich odwołać w pliku *main.qml*. W ten sposób odizolujemy definicję wyglądu aplikacji od jej funkcjonalności.

Druga strona aplikacji została pokazana na **listingu 3**. W pliku *Page2Form.ui.qml* importujemy moduł *QtCharts* i dodajemy strukturę *ChartView*, w której definiujemy wykres liniowy zawierający sto punktów. W strukturze *LineSeries* dodajemy dwie osie danych. Oś Y będzie zawierała losowo generowane wartości od 0 do 100, natomiast na osi X, której wartości są ukryte będziemy dodawać kolejne numery punktów. Początkowa minimalna wartość na osi X jest ustawiona na –100, dzięki czemu punkty na wykresie będą pojawiały się od prawej strony okna. Na tej stronie również dodajemy aliasy właściwości, do których będziemy odwoływać się w pliku *main.qml*.

Opis głównego okna aplikacji dodajemy w pliku *main.qml*, tak jak na **listingu 4**. Zamiast definiowania szerokości i wysokości okna definiujemy widok pełnoekranowy, a następnie dodajemy funkcje do kontrolek oraz wykresu. Do przycisku i suwaka dodajemy wywołania funkcji obiektu *ledHandler*, który dodamy za chwilę do projektu, a który będzie odpowiedzialny za włączanie i ustawianie jasności diod. W kodzie drugiej strony dodajemy powiązanie metody *onDoAddSample* obiektu *pointGenerator*, który również dodamy za chwilę do projektu, z funkcją *addPoint* zdefiniowaną poniżej. Funkcja ta dodaje punkt do wykresu pilnując jednocześnie, aby liczba punktów nie przekroczyła wartości *maxNumberOfPoints*. Na końcu pliku znajdują się pozostawione bez zmian definicje przycisków umożliwiających przełączanie się pomiędzy stronami.

Teraz przejdziemy do implementacji klas *LedHandler* i *PointGenerator* odpowiedzialnych odpowiednio za obsługę diod oraz generację punktów dla wykresu. Nową klasę dodajemy klikając prawym przyciskiem myszy na nazwę projektu po lewej stronie

Listing 5. Plik nagłówkowy klasy *LedHandler*

```
#ifndef LEDHANDLER_H
#define LEDHANDLER_H

#include <QObject>
#include <QFile>

class LedHandler : public QObject{
    Q_OBJECT
    public:
        explicit LedHandler(QObject *parent = nullptr);

    signals:

    public slots:
        void setEnabled(bool enabled);
        void setBrightness(unsigned int brightness);

    private:
        QFile* gpioLedFile;
        QFile* pwmLedFile;
};

#endif // LEDHANDLER_H
```

Listing 7. Plik nagłówkowy klasy *PointGenerator*

```
#ifndef POINTGENERATOR_H
#define POINTGENERATOR_H

#include <QObject>
#include <QTimer>

class PointGenerator : public QObject{
    Q_OBJECT
    public:
        explicit PointGenerator(QObject *parent = nullptr);

    signals:
        void doAddSample(int value);

    public slots:
        void timeout();

    private:
        QTimer* timer;
};

#endif // POINTGENERATOR_H
```

Listing 6. Plik źródłowy klasy *LedHandler*

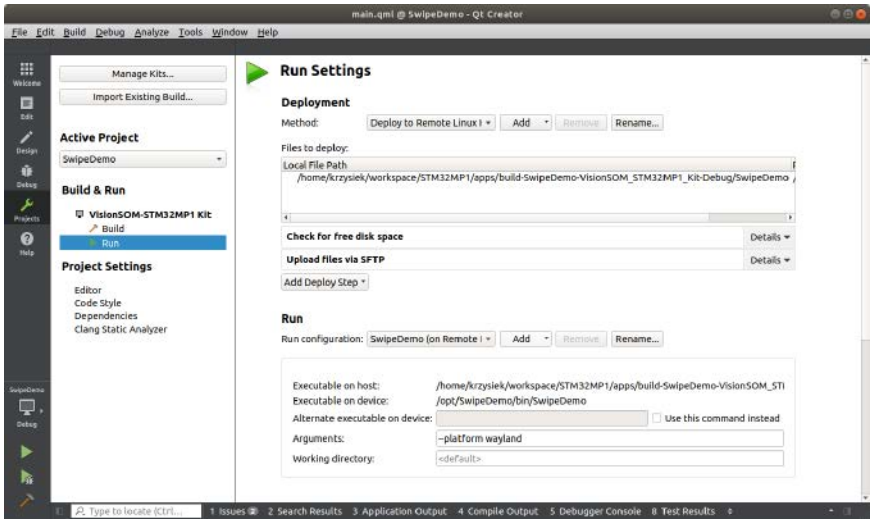
```
#include "ledhandler.h"
#include <QDebug>

LedHandler::LedHandler(QObject *parent) : QObject(parent){
    this->gpioLedFile = new QFile("/sys/class/leds/led3/brightness");
    this->gpioLedFile->open(QIODevice::WriteOnly);
    this->pwmLedFile = new QFile("/sys/class/leds/led2/brightness");
    this->pwmLedFile->open(QIODevice::WriteOnly);
}

void LedHandler::setEnabled(bool enabled){
    qDebug() << "Enabled" << enabled;

    if (enabled)
        this->gpioLedFile->write("1");
    else
        this->gpioLedFile->write("0");

    this->gpioLedFile->flush();
}

void LedHandler::setBrightness(unsigned int brightness){
    qDebug() << "Brightness" << brightness;
    this->pwmLedFile->write(QString::number(brightness).toLatin1());
    this->pwmLedFile->flush();
}
```

głównego okna IDE, a z dostępnych szablonów wybieramy C++ Class. Następnie w kreatorze podajemy nazwę klasy i klasę bazową, którą powinna być klasa QObject, tak jak na **rysunku 8.**

Po zakończeniu kreatora do projektu zostaną dodane dwa pliki nowej klasy: nagłówkowy i źródłowy. Uzupełniamy je zgodnie z kodem podanym na **listingu 5 i listingu 6.** Obsługa diod będzie wykonywana za pośrednictwem plików znajdujących się w katalogu systemowym */sys/class/leds*, dlatego do pliku nagłówkowego dodajemy dwa pola typu *QFile** reprezentujące dwie diody obsługiwane przez przycisk (*gpioLedFile*) i suwak (*pwmLedFile*). Dodajemy także dwie metody wywoływane w efekcie zmiany stanu komponentów graficznych. Wykorzystują one mechanizm Signals & Slots biblioteki Qt, dzięki czemu możemy powiązać metody deklarowane jako slots, z sygnałami wysyłanymi przez inne obiekty. Więcej na ten temat można przeczytać w oficjalnej dokumentacji Qt.

W pliku źródłowym dodajemy implementację konstruktora klasy, w którym tworzymy obiekty plików diod i otwieramy je do zapisu. Dodajemy także metody *setEnabled*, która zapisuje do pliku "1" lub "0", w zależności od nowego stanu przycisku, oraz *setBrightness*, która wpisuje do pliku wartość reprezentującą jasność diody. W obu przypadkach wywołujemy funkcję *flush*, która ma za zadanie opróżnić bufor, dzięki czemu zapis do plików następuje od razu po wywołaniu wspomnianych metod. W kodzie źródłowym możemy także użyć klasy *qDebug*, która pozwala nam na wysyłanie komunikatów diagnostycznych na standardowe wyjście aplikacji.

W ten sam sposób dodajemy klasę *PointGenerator*, której nagłówek i plik źródłowy znajdują się na **listingu 7** i **listingu 8**. W pliku nagłówkowym dodajemy timer pozwalający nam na cykliczne dodawanie punktów do wykresu. Wymaga on implementacji metody wywoływanej po ustalonym czasie – w naszym przykładzie jest to *timeout*. Deklarujemy tu również sygnał *doAddSample*, który powiązaliśmy z dodawaniem punktów wykresu w pliku *main.qml*. W pliku *cpp* dodajemy implementację konstruktora, gdzie używając funkcji *connect* tworzymy i uruchamiamy timer łącząc jednocześnie jego sygnał *timeout* z metodą naszej klasy o tej samej



Rysunek 9. Dodanie parametru *platform* do konfiguracji projektu

Rysunek 10. Strona aplikacji zawierająca wykres

Rysunek 11. Pierwsza strona aplikacji z ustawionym stylem *Material Dark*

Listing 8. Plik źródłowy klasy *PointGenerator*

```
#include "pointgenerator.h"
#include <QDateTime>

PointGenerator::PointGenerator(QObject *parent) : QObject(parent){
    this->timer = new QTimer();
    this->timer->setSingleShot(false);
    this->timer->start(1000);
    connect(this->timer, SIGNAL(timeout()), this, SLOT(timeout()));
    qsrand(QDateTime::currentMSecsSinceEpoch() / 1000);
}

void PointGenerator::timeout(){
    emit this->doAddSample(qrand() % 100);
}
```

Listing 9. Plik *main.cpp*

```
#include <QApplication>
#include <QQmlApplicationEngine>
#include <QQmlContext>
#include "ledhandler.h"
#include "pointgenerator.h"

int main(int argc, char *argv[]){
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QApplication app(argc, argv);

    LedHandler ledHandler;
    ledHandler.setBrightness(0);
    ledHandler.setEnabled(false);

    PointGenerator pointGenerator;
    QQmlApplicationEngine engine;

    engine.rootContext()->setContextProperty("ledHandler", &ledHandler);
    engine.rootContext()->setContextProperty("chartBackend", &pointGenerator);

    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```

nazwie. Na końcu kodu konstruktora inicjalizujemy generator liczb pseudolosowych za pomocą aktualnego czasu. W metodzie *timeout* wysyłamy jedynie sygnał *doAddSample*, którego argumentem jest wygenerowana liczba pseudolosowa.

Mając gotowe wszystkie komponenty możemy zmodyfikować plik *main.cpp* pokazany na **listingu 9**. Ze względu na użycie modułu *QtCharts* musimy zmienić typ aplikacji z QGuiApplication na QApplication dodając również odpowiedni nagłówek. Następnie tworzymy obiekty dodanych przez nas klas – *ledHandler* oraz *pointGenerator* oraz ustawiamy jasność obu diod na 0. Utworzone obiekty łączymy z kodem QML za pomocą metody *setContextProperty*, w której podajemy nazwę używaną w plikach qml oraz referencję do odpowiedniego obiektu implementującego wymaganą logikę.

Pozostało nam jeszcze dodanie odpowiednich modułów w pliku projektu *SwipeDemo.pro*. Do zmiennej QT dodajemy moduły *qml* i *charts*:

```
QT += quick qml charts
```

Nasza aplikacja jest już gotowa do uruchomienia na platformie docelowej. Do poprawnego działania musimy jednak podać odpowiedni parametr *--platform*, dzięki któremu zostaną załadowane dostępne na urządzeniu moduły. Klikamy więc na przycisk Projects po lewej stronie okna głównego IDE i w oknie Run Settings wpisujemy do pola Arguments: `--platform wayland`. Zostało to pokazane na **rysunku 9**. Po kliknięciu na zieloną strzałkę w lewym dolnym rogu okna aplikacja zostanie skopiowana na urządzenie i uruchomiona. Na **rysunku 10** został pokazany widok strony z wykresem.

Styl aplikacji

Biblioteka Qt umożliwia nam w bardzo łatwy sposób zmianę wyglądu aplikacji na jeden z predefiniowanych styli. Służy do tego plik *qtquickcontrols2.conf*, w którym domyślny wygląd możemy zmienić, np. na *Material Dark* tak jak na **listingu 10**. Po tej zmianie otrzymamy efekt, taki jak na **rysunku 11**.

Listing 10. Styl aplikacji zmieniony na *Material Dark*

```
[Controls]
Style=Material

[Material]
Theme=Dark
```

Listing 11. Sekcja w pliku *weston.ini* dodająca skrót do aplikacji na pasku pulpitu

```
[launcher]
icon=/usr/share/weston/icon_window.png
path=/opt/SwipeDemo/bin/SwipeDemo --platform wayland
```

Skrót do aplikacji

Na koniec zobaczymy w jaki sposób dodać skrót aplikacji do paska znajdującego się na pulpicie domyślnego systemu graficznego OpenSTLinuxa, czyli *Wayland/Weston*. Będziemy do tego potrzebować pliku binarnego aplikacji i pliku graficznego ikony. Jeżeli wcześniej uruchomiliśmy aplikację z poziomu Qt Creatora, to plik binarny znajdziemy w katalogu */opt/SwipeDemo/bin/*. Jako ikony użyjemy z kolei pliku */usr/share/weston/icon_window.png* znajdującego się w systemie plików domyślnego obrazu systemu. Teraz możemy otworzyć plik */etc/xdg/weston/weston.in*i za pomocą dowolnego edytora tekstu znajdującego się na urządzeniu, np. vi, i dodać sekcję pokazaną na **listingu 11**. Po ponownym uruchomieniu systemu, na pasku na pulpicie pojawi się nowa ikona, a po jej wciśnięciu zostanie uruchomiona nasza aplikacja.

Krzysztof Chojnowski