APPENDIX D

Class library for Sense HAT sensors

In this appendix, I show you how to develop the Portable Class Library (PCL) and reference it in other projects. The idea is to build a reusable PCL that can be shared across various projects. You can even reuse the common C# code between different platforms.

You use the I²C interface to associate communication with all the Sense HAT sensors. To read and write data through this interface, UWP implements the `I2cDevice` class. Given the raw bytes obtained from a sensor, you then convert them to meaningful values. The conversion logic is platform-independent, contrary to the `I2cDevice` object, which is inherent to UWP. Hence, the conversion logic could be used on different platforms, which most likely use objects other than an `I2cDevice` to handle I²C communication.

Isolating platform-independent from platform-dependent code is a good practice. Cross-platform programming uses that approach extensively. Here, I divide the code of the SenseHat project (companion code: Chapter 05/SenseHat) into three separate projects. As shown in Figure D-1, the solution has a hierarchical structure:

- The bottom layer consists of the PCL, implementing platform-independent helpers and conversion logic.
- Above the PCL, I have the UWP Class Library—converters, I²C-related logic, and additional helpers, which use objects from the UWP.
- On the top of this stack, I have the main UWP app, which uses the code from the PCL and UWP class library.

You can find the whole implementation in the companion code under the Appendix D folder.

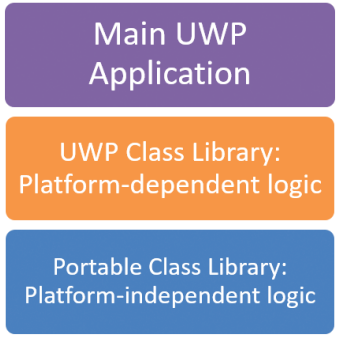

FIGURE D-1 A structure of a typical project utilizing the Portable Class Library to isolate platform-independent logic.

Portable Class Library

Start by creating the PCL project as follows:

1. Open the New Project window.
2. In the search text box, type **Class Library Portable**.
3. Choose the **Class Library (Portable) Visual C#** project template.
4. Set the project name to **SenseHat.Portable** and the solution name to **SenseHat**. (See Figure D-2.) Then click **OK**.

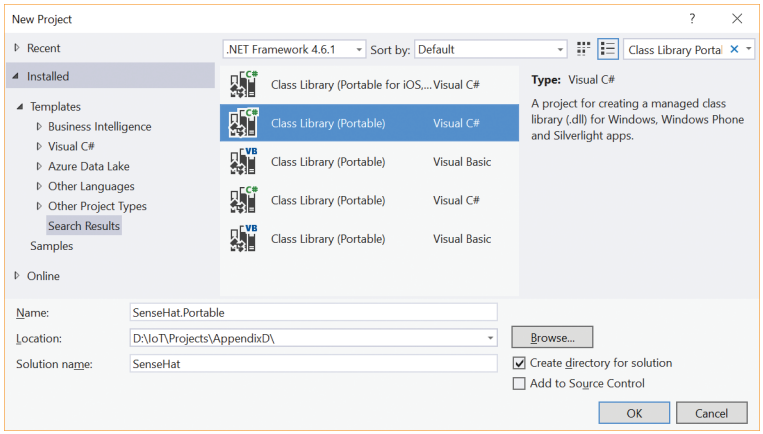

FIGURE D-2 The New Project dialog box for creating the SenseHat.Portable PCL.

A new dialog box, Add Portable Class Library, appears. This dialog box allows you to choose target platforms supported by the SenseHat.Portable PCL. If the Add Portable Class Library dialog box does not appear, go to the SenseHat.Portable properties. Then, on the Library tab, click the **Change** button. It activates the Change Targets dialog box (see Figure D-3), which looks the same as the Add Portable Class Library dialog box. They differ by the caption only.

Using either the Add Portable Class Library or Change Targets dialog box, choose the following platforms:

- .NET Framework 4.5
- Windows 8
- Windows Phone Silverlight 8
- ASP.NET Core 1.0
- Windows Phone 8.1

The PCL will be compatible with these platforms. You can always modify your choice by reopening the Change Targets dialog box.

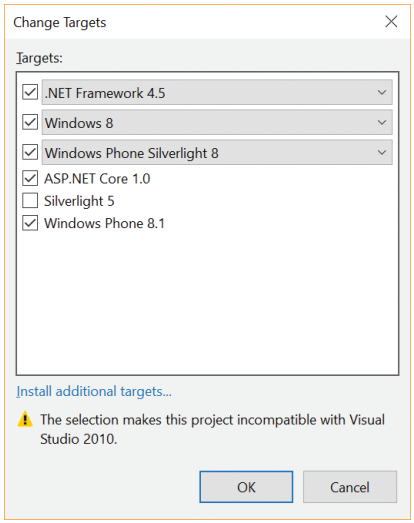

FIGURE D-3 The Change Targets dialog box lets you configure target platforms.

Under the Sense Hat.Portable project, create the Helpers folder (right-click on **SenseHat.Portable**, choose **Add**, and select **New Folder**). Then open the same menu, choose **Add**, and select **Existing Item** to add to this folder the following files from the SenseHat project (see the companion code in Chapter 05/SenseHat/Helpers):

- Check.cs
- Constants.cs
- ConversionHelper.cs
- HumiditySensorHelper.cs
- InertialSensorHelper.cs
- MagneticFieldSensorHelper.cs
- TemperatureAndPressureSensorHelper.cs
- Vector3D.cs

Finally, change the namespace used in every file from `SenseHat.Helpers` to `SenseHat.Portable.Helpers`.

Universal Windows Class Library

Given the PCL, you can now build the UWP class library by performing the following steps:

1. In the Solution Explorer, right-click **Solution 'SenseHat'** and choose **Add/New Project** from the context menu.
2. In the search box of the Add New Project dialog box, type **Class Library Universal**.

3. Choose the **Class Library (Universal Windows) Visual C#** project template and change its name to **SenseHat.UWP**.

4. Set the target and minimum version to **Windows 10 Anniversary Edition (10.0; Build 14393)** and **Windows 10 (10.0; Build 10586)**, respectively.

Now, reference the PCL in the SenseHat.UWP project as follows:

1. In the Solution Explorer, right-click the **References** node of the SenseHat.UWP project and select **Add Reference** from the context menu. The Reference Manager dialog box appears.

2. In the Reference Manager dialog box, navigate to **Projects/Solution** and choose **SenseHat.Portable**. (See Figure D-4.)

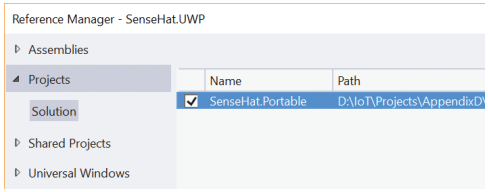

FIGURE D-4 Referencing SenseHat.Portable PCL in the SenseHat.UWP project.

3. Finally, reference the Windows IoT Extensions for the UWP (version 14393) and close the Reference Manager dialog box.

Under the SenseHat.UWP Class Library, I create three folders—Converters, Helpers, and Sensors—and add appropriate files from the corresponding folders of the SenseHat app from Chapter 5, "Reading data from sensors." So, the SenseHat.UWP project has the structure depicted in Figure D-5. Note that SenseHat.UWP contains all the files that use the `I2cDevice` class. This object is unavailable in platforms that are targeted by SenseHat.Portable.

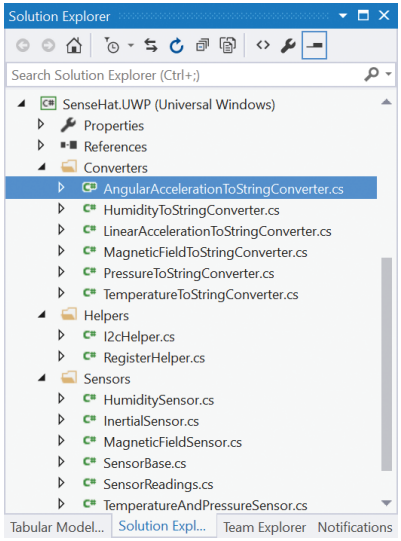

FIGURE D-5 The structure of the SenseHat.UWP project.

I also modify the namespaces by adding the UWP acronym:

- `SenseHat.Converters` becomes `SenseHat.UWP.Converters`
- `SenseHat.Helpers` becomes `SenseHat.UWP.Helpers`
- `SenseHat.Sensors` becomes `SenseHat.UWP.Sensors`

Note that after you change the namespaces, you also need to update the corresponding `using` statements. You can do this automatically by using the Visual Studio refactoring module. It activates a hint (yellow light bulb) whenever you change the name of variable, class, struct, or namespace.

Universal Windows Platform application

On top of the preceding class libraries, I create the main application, which displays sensor readings from the Sense HAT add-on board. I first supplement the SenseHat solution with the new project, SenseHat.Sensors, created by using the Blank App (Universal Windows) Visual C# project template. I set the target and minimum to the SenseHat.UWP Class Library. Subsequently, I reference the SenseHat.Portable and SenseHat.UWP projects, and set the SenseHat.Sensors as the startup project (right-click **SenseHat.Sensors** and choose **Set as StartUp Project**).

Before going further, let's inspect the project dependencies and build order. You can visualize them by using a dedicated dialog box. You activate this dialog box in the Solution Explorer by right-clicking the **SenseHat** solution and choosing **Project Dependencies** from the context menu. Portions of the Project Dependencies dialog box appear in Figure D-6. This dialog box includes two tabs: Dependencies and Build Order. Use the first tab to review the hierarchy of your projects and the second to see how your solution will be built.

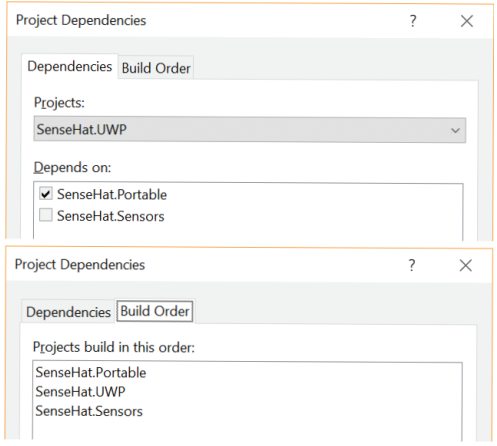

FIGURE D-6 Project dependencies and build order of the SenseHat solution.

Figure D-6 shows that the SenseHat.Portable project will be compiled first, and then the compiler will build the SenseHat.UWP project, and finally the SenseHat.Sensors project. It follows from the fact that the SenseHat.Portable is referenced by the SenseHat.UWP, which is used in the SenseHat.Sensors app.

The SenseHat.Sensors app was implemented in a similar way as the SenseHatTelemeter app (see Chapter 12, "Remote device monitoring"). Namely, to periodically read data from sensors, I implement the `Telemetry` and `TelemetryEvent-Args` classes. Both are built by extending classes from `SenseHat-Telemeter` to include pressure, linear and angular acceleration, and magnetic field sensor readings.

The `Telemetry` class (see the companion code at Appendix D/SenseHat.Sensors/TelemetryControl/Telemetry.cs) does not implement any public constructors. Instead, it has the static asynchronous factory method `CreateAsync`, which appears in Listing D-1.

LISTING D-1 Asynchronous factory method of the Telemetry class

```
public static async Task<Telemetry> CreateAsync(TimeSpan readoutDelay)
{
    Check.IsNull(readoutDelay);

    var telemetry = new Telemetry(readoutDelay);

    await telemetry.InitializeSensors();

    return telemetry;
}

private TimeSpan readoutDelay;

private Telemetry(TimeSpan readoutDelay)
{
    this.readoutDelay = readoutDelay;
}
```

By inspecting the code snippet from Listing D-1, you see that because I initialize sensors there, the `Telemetry` class must be created by using the asynchronous static factory method. As you know from Chapter 5, sensor classes use asynchronous code. However, in C#, constructors cannot be asynchronous, so I use the static asynchronous factory method.

The `InitializeSensors` private method appears in Listing D-2. As you can see, I invoke the `Initialize` method of each class, representing a sensor. Then, I verify that the particular class was initialized using the `VerifyInitialization` method (bottom part of Listing D-2). If it turns out that a sensor is unavailable (was not initialized), I throw an exception with an appropriate message. This informs the caller that the `Telemetry` class was unable to access all required sensors and cannot proceed further.

LISTING D-2 Sensors initialization

```
private TemperatureAndPressureSensor temperatureAndPressureSensor =
    TemperatureAndPressureSensor.Instance;
private HumidityAndTemperatureSensor humidityAndTemperatureSensor =
    HumidityAndTemperatureSensor.Instance;
private InertialSensor inertialSensor = InertialSensor.Instance;
private MagneticFieldSensor magneticFieldSensor = MagneticFieldSensor.Instance;
```

```
private async Task InitializeSensors()
{
    await temperatureAndPressureSensor.Initialize();
    VerifyInitialization(temperatureAndPressureSensor,
        "Temperature and pressure sensor is unavailable");

    await humidityAndTemperatureSensor.Initialize();
    VerifyInitialization(humidityAndTemperatureSensor,
        "Humidity sensor is unavailable");

    await inertialSensor.Initialize();
    VerifyInitialization(inertialSensor, "Inertial sensor is unavailable");

    await magneticFieldSensor.Initialize();
    VerifyInitialization(magneticFieldSensor, "Magnetic field sensor is unavailable");
}

private void VerifyInitialization(SensorBase sensorBase, string exceptionMessage)
{
    if(!sensorBase.IsInitialized)
    {
        throw new Exception(exceptionMessage);
    }
}
```

After a successful initialization, start a periodic background operation of sensor readings by using the `Start` method of the `Telemetry` class instance (see Listing D-3). This method first checks whether the telemetry is active; if not, it initializes and then starts the `Telemetry` worker thread.

LISTING D-3 Starting telemetry task

```
public bool IsActive { get; private set; } = false;

public void Start()
{
    if (!IsActive)
    {
        InitializeTelemetryTask();

        telemetryTask.Start();

        IsActive = true;
    }
}
```

The `Telemetry` worker thread is configured within `InitializeTelemetryTask` (see Listing D-4). This method creates a `Task` instance, which is implemented to periodically take sensor readings and report them to listeners through a `DataReady` event. Under the `while` loop from Listing D-4, I also check whether the `IsActive` flag is `true` before raising an event.

LISTING D-4 Initializing telemetry task

```
private Task telemetryTask;
private CancellationTokenSource telemetryCancellationTokenSource;

private void InitializeTelemetryTask()
{
    telemetryCancellationTokenSource = new CancellationTokenSource();

    telemetryTask = new Task(() =>
    {
        while (!telemetryCancellationTokenSource.IsCancellationRequested)
        {
            if (IsActive)
            {
                var telemetryEventArgs = GetSensorReadings();

                DataReady(this, telemetryEventArgs);

                Task.Delay(readoutDelay).Wait();
            }
        }
    }, telemetryCancellationTokenSource.Token);
}
```

As in the SenseHatTelemeter app, the worker thread (background operation) can be stopped through a dedicated `CancellationTokenSource` (the `Stop` method from Listing D-5).

LISTING D-5 Stopping method

```
public void Stop()
{
    if (IsActive)
    {
        telemetryCancellationTokenSource.Cancel();

        IsActive = false;
    }
}
```

When the worker thread is active, sensor readings are obtained by using the `GetSensorReadings` method from Listing D-6. This function sequentially invokes dedicated methods of classes representing sensors, e.g. `temperatureAndPressureSensor.GetTemperature`, `inertialSensor.GetLinearAcceleration`, and so on. The resulting values are wrapped into an instance of the `TelemetryEventArgs` object (see Listing D-7), which is then passed to listeners.

LISTING D-6 Reading values from sensors

```
private TelemetryEventArgs GetSensorReadings()
{
    var temperature = temperatureAndPressureSensor.GetTemperature();
    var humidity = humidityAndTemperatureSensor.GetHumidity();
    var pressure = temperatureAndPressureSensor.GetPressure();

    var linearAcc = inertialSensor.GetLinearAcceleration();
    var angularSpeed = inertialSensor.GetAngularSpeed();
    var magneticField = magneticFieldSensor.GetMagneticField();

    return new TelemetryEventArgs(temperature, humidity, pressure,
        linearAcc, angularSpeed, magneticField);
}

public class TelemetryEventArgs
{
    public float Temperature { get; private set; }

    public float Humidity { get; private set; }

    public float Pressure { get; private set; }

    public Vector3D<float> LinearAcceleration { get; private set; }

    public Vector3D<float> AngularSpeed { get; private set; }

    public Vector3D<float> MagneticField { get; private set; }

    public TelemetryEventArgs(float temperature, float humidity, float pressure,
        Vector3D<float> linearAcc, Vector3D<float> angularSpeed,
        Vector3D<float> magneticField)
    {
        Temperature = temperature;
        Humidity = humidity;
        Pressure = pressure;

        LinearAcceleration = linearAcc;
        AngularSpeed = angularSpeed;
        MagneticField = magneticField;
    }
}
```

Figure D-7 depicts the UI of SenseHat.Sensors, when the app is executed on the desktop platform. The UI is composed of three tabs: Control, Weather, and Inertial. The first tab has three buttons, which initialize sensors and start and stop telemetry. The Weather and Inertial tabs present values from sensors. The Weather tab shows temperature, humidity, and pressure, while the Inertial tab displays linear acceleration (from accelerometer) and angular speed (from gyroscope) and the magnetic field (from magnetometer).

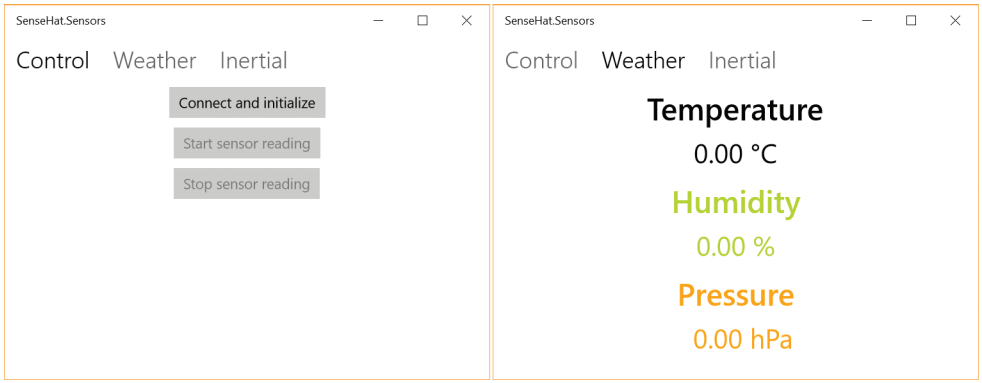

FIGURE D-7 The two first tabs of the SenseHat.Sensors UI, when run on the desktop platform.

To associate a connection with the Sense HAT add-on board and initialize sensor classes, you click the Connect and Initialize button. Doing so invokes the `ButtonConnect_Click` event handler from Listing D-7. This method checks whether sensors are already initialized and then creates a `Telemetry` object by invoking the `Telemetry.CreateAsync` method. Subsequently, the handler is attached to the `DataReady` event, which is raised whenever new sensor readings are available. Note that a method from Listing D-7 also configures properties of the `SensorsViewModel` class instance, which I tell you about later in this section.

LISTING D-7 Sensors initialization

```
private Telemetry telemetry;

private SensorsViewModel sensorsViewModel = new SensorsViewModel()
{
    ReadoutDelay = TimeSpan.FromSeconds(1)
};

private async void ButtonConnect_Click(object sender, RoutedEventArgs e)
{
    if (!sensorsViewModel.IsConnected)
    {
        try
        {
            telemetry = await Telemetry.CreateAsync(sensorsViewModel.ReadoutDelay);
            telemetry.DataReady += Telemetry_DataReady;

            sensorsViewModel.IsConnected = true;
        }
        catch(Exception ex)
        {
            Debug.WriteLine(ex.Message);
        }
    }
}
```

Within the event handler of the DataReady event, shown in Listing D-8, I use an instance of TelemetryEventArgs to display sensor readings in the UI. I do this indirectly by updating corresponding properties of the SensorsViewModel class. Since the DataReady event is raised from the worker thread, I need to dispatch property rewriting to the UI thread.

LISTING D-8 Displaying sensor readings

```
private void Telemetry_DataReady(object sender, TelemetryEventArgs e)
{
    DisplaySensorReadings(e);
}

private async void DisplaySensorReadings(TelemetryEventArgs telemetryEventArgs)
{
    await Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        sensorsViewModel.SensorReadings.Temperature = telemetryEventArgs.Temperature;
        sensorsViewModel.SensorReadings.Humidity = telemetryEventArgs.Humidity;
        sensorsViewModel.SensorReadings.Pressure = telemetryEventArgs.Pressure;

        sensorsViewModel.SensorReadings.Accelerometer = telemetryEventArgs.
            LinearAcceleration;
        sensorsViewModel.SensorReadings.Gyroscope = telemetryEventArgs.AngularSpeed;
        sensorsViewModel.SensorReadings.Magnetometer = telemetryEventArgs.MagneticField;
    });
}
```

Listing D-9 shows event handlers of Start and Stop sensor readings. These simply invoke corresponding methods of the Telemetry class instance and update button status through SensorsViewModel properties.

LISTING D-9 Starting and stopping telemetry

```
private void ButtonStartSensorReading_Click(object sender, RoutedEventArgs e)
{
    telemetry.Start();
    sensorsViewModel.IsTelemetryActive = telemetry.IsActive;
}

private void ButtonStopSensorReading_Click(object sender, RoutedEventArgs e)
{
    telemetry.Stop();
    sensorsViewModel.IsTelemetryActive = telemetry.IsActive;
}
```

SensorsViewModel controls the state and updates the UI (see the companion code in Appendix D/SenseHat.Sensors/MainPage.xaml). The main elements of SensorsViewModel are the following six public properties (see the companion code in Appendix D/SenseHat.Sensors/ViewModels/SensorsViewModel.cs):

- **`SensorReadings`** This stores values obtained from sensors. Members of the `SensorReadings` class are one-way bound to the UI. Whenever you change the `SensorReadings` property of `SensorsViewModel`, the UI is updated.
- **`ReadoutDelay`** This specifies a delay between consecutive sensor readings. Its default value, configured in Listing D-7, is 1 second.
- **`IsConnected`** This indicates whether the `Telemetry` class was initialized. If so, the Start Sensor Reading button can be enabled.
- **`IsTelemetryActive`** This is a flag that informs the UI whether the background operation of sensor reading is active. If so, the Start Sensor Reading button is enabled, and the Stop Sensor Reading Button is disabled. Otherwise, the `Enabled` property of these buttons is reversed.
- **`IsStartSensorReadingButtonEnabled`** This controls the `Enabled` property of the Start Sensor Reading button.
- **`IsStopSensorReadingButtonEnabled`** This controls the `Enabled` property of the Stop Sensor Reading button.

When you run the app, you can click the Connect and Initialize button. If the Sense HAT add-on board is present, and everything went correctly, the Start Sensor Reading button will be enabled. If you click it, the background operation starts, and actual sensor readings are displayed in the UI until you click the Stop Sensor Reading button. The sample readings from the inertial sensor are given in Figure D-8.

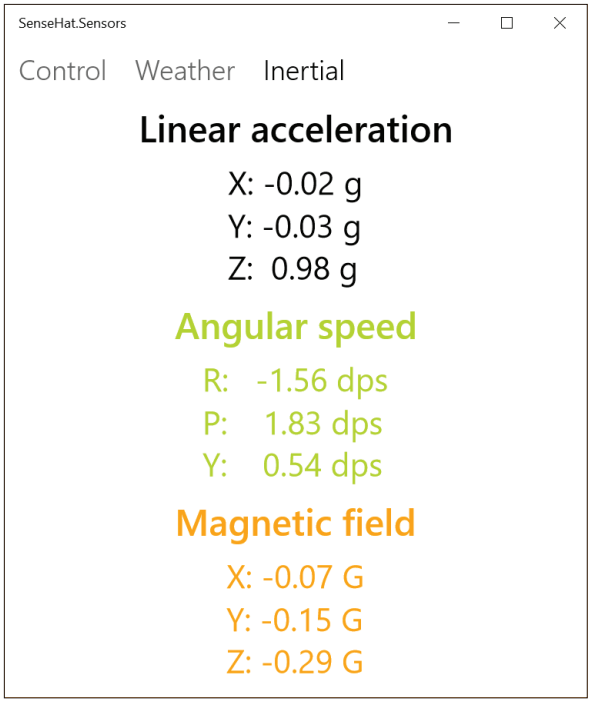

FIGURE D-8 The Inertial tab of the SenseHat.Sensors app.