APPENDIX E

Visual C++ component extensions

Visual C++ component extensions (C++/CX) constitute the set of language extensions that enable you to directly interface Windows Runtime and native C/C++ code. As I showed you in Chapter 8, "Image processing," you can build such C++/CX components for complex operations (like image processing) and to reuse existing code and libraries. Typically, you create C++/CX component as a Windows Runtime component or a DLL, which is then referenced by the UWP project. In that project, you invoke methods from the C++/CX component in exactly the same manner as you would with any other UWP assembly. In this way, C++/CX provides an access to native programming, while giving you an opportunity to use modern elements of C++ programming and straightforward integration with higher layers of your UWP solution.

You can also write the entire UWP app in C++/CX. This can be particularly useful when you prefer C++ over C# or need improved speed performance since C++/CX app compiles to native code regardless the build configuration. However, in most cases you will most likely prepare a C++/CX wrapper to interface legacy code, and then write the UI layer and accompanied logic with C#.

In this appendix I show you how to start with building a UWP app with C++/CX. In particular, I tell you how to implement an app that will emulate periodic sensor readings in the background. Such background operation was one of the most common functionalities I dealt with throughout this book. This app will illustrate the most important elements of C++/CX from the IoT perspective. Further details and the full C++/CX language reference can be found in the following guide: *https://bit.ly/cpp_cx*.

User interface and event handling

I attached the source code supporting this discussion to the companion code under the Appendix E subfolder. I created one project, SenseHat.Sensors.CppCx, using the Blank App (Universal Windows) Visual C++ project template. Then I built a UI consisting of two buttons and two labels (see Figure E-1). Again, buttons are used to start and stop background operations. One label displays the constant string `Temperature`, and the other shows the emulated sensor reading.

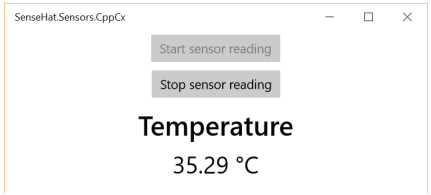

FIGURE E-1 UI of SenseHat.Sensors.CppCx.

UI declaration uses the same XAML syntax as the C# app. (See the companion code at Appendix E/MainPage.xaml.) However, the associated code-behind is composed of two files: MainPage.xaml.h and MainPage.xaml.cpp. The first one contains the `MainPage` class declaration and the other stores its definition.

As shown in Listing E-1, I extend the `MainPage` declaration by one public read-only property (`ViewModel` of type `SensorsViewModel^`), two private fields (`sensorsViewModel` and `telemetry`), and also three methods (`OnDataReady`, `ButtonStartSensorReading_Click`, and `ButtonStopSensorReading_Click`). Analyzing these declarations, the most conspicuous part of the syntax is ^ (hat). This declarator acts as a smart pointer. It instructs the runtime to automatically manage the lifetime of the object, so corresponding memory will be released when the reference counter goes to zero. Additionally, hats provide automatic type conversion. Therefore, types with hats are typically referred to as *managed types*.

LISTING E-1 MainPage class declaration

```
public ref class MainPage sealed
{
public:
    MainPage();

    property SensorsViewModel^ ViewModel
    {
        SensorsViewModel^ get() { return sensorsViewModel; }
    }

private:
    SensorsViewModel^ sensorsViewModel = ref new SensorsViewModel();
    Telemetry^ telemetry = ref new Telemetry();

    void OnDataReady(Object^ sender, TelemetryEventArgs^ e);
    void ButtonStartSensorReading_Click(Object^ sender, RoutedEventArgs^ e);
    void ButtonStopSensorReading_Click(Object^ sender, RoutedEventArgs^ e);
};
```

In `MainPage`, most types are declared with hats because they are related to the UWP API. However, as you will shortly see, in C++/CX you can also use standard pointers, declared with *. You construct managed types with the `ref new` keyword, while for unmanaged types you typically use the `new` keyword. To access members of the managed type, you use the member-access operator, `->`.

Definitions of the button event handlers appear in Listing E-2. They start and stop telemetry and update the `IsTelemetryActive` property of `ViewModel`. The latter is bound to the UI to control the state of buttons and display the emulated temperature.

As shown in Listing E-3, the temperature value is reported using a `DataReady` event of the `Telemetry` class. The code block from Listing E-3 also shows how to associate an event handler with the event. This is done within the `MainPage` constructor, where using the += operator I create the event handler of the `DataReadyEventHandler` delegate type. The statement looks very similar to its C# counterpart. However, to indicate the actual method that handles an event, you use the & operator.

LISTING E-2 Starting and stopping telemetry

```
void MainPage::ButtonStartSensorReading_Click(Object^ sender, RoutedEventArgs^ e)
{
   telemetry->Start();
   ViewModel->IsTelemetryActive = telemetry->IsActive;
}

void MainPage::ButtonStopSensorReading_Click(Object^ sender, RoutedEventArgs^ e)
{
   telemetry->Stop();
   ViewModel->IsTelemetryActive = telemetry->IsActive;
}
```

LISTING E-3 Event handling

```
MainPage::MainPage()
{
    InitializeComponent();
    telemetry->DataReady += ref new DataReadyEventHandler(this, &MainPage::OnDataReady);
}

void MainPage::OnDataReady(Object^ sender, TelemetryEventArgs^ e)
{
    if (Dispatcher->HasThreadAccess)
    {
        ViewModel->Temperature = e->Temperature;
    }
    else
    {
        Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
            ref new DispatchedHandler([=]()
        {
            MainPage::OnDataReady(sender, e);
        }));
    }
}
```

In this example, an event is consumed in the `OnDataReady` method. As shown in Listing E-3, this method rewrites the temperature value to the corresponding property of `ViewModel`. Since `ViewModel` is bound to UI, it will automatically trigger a UI update, so I have to ensure that this operation would be thread-safe. I use the `Dispatcher` object, which is employed similarly as in C# projects. Only one aspect requires additional comments. That is the statement where I create `DispatchedHandler`. There I use a lambda expression (or simply lambda; *https://bit.ly/cpp_lambda*) with the capture clause `[=]`. This clause is used to capture variables from the scope surrounding the lambda expression to the lambda body. In that case, all variables will be captured by value. If you need to capture variables by reference, then you use `[&]` clause. You can also use specific capture types for particular variables. Simply write variable names within the capture clause, and precede them by & for reference capture. An empty clause [] will not capture any variables.

Event declaration and event arguments

The declaration of `DataReadyEventHandler`, which I used previously, is saved in the Telemetry header file. (See the companion code at Appendix E/TelemetryControl/Telemetry.h.) As shown in Listing E-4, `DataReadyEventHandler` contains the access modifier, `delegate` keyword, return type, and the list of formal arguments. Apart from the appearance of hats, the declaration looks exactly the same as in C#.

LISTING E-4 Delegate type declaration

```
public delegate void DataReadyEventHandler(Object^ sender, TelemetryEventArgs^ e);
```

`DataReadyEventHandler` has two arguments: `sender` and `e`. The first is used to pass the reference to an object, raising an event, while the latter stores the telemetry data. In this case, the sensor readings are wrapped in the `TelemetryEventArgs` class (see the companion code at Appendix E/TelemetryControl), whose declaration appears in Listing E-5.

LISTING E-5 Declaration of the TelemetryEventArgs class

```
public ref class TelemetryEventArgs sealed
{
public:
    TelemetryEventArgs(double temperature);

    property double Temperature
    {
        public: double get() { return temperature; }
        private: void set(double value) { temperature = value; }
    }

private:
    double temperature;
};
```

`TelemetryEventArgs` declares one private field, `temperature`, and one public property, `Temperature`. According to the property declaration, the `Temperature` member can be accessed publicly but modified privately. We see that in this case, C# syntax would be much more straightforward for accomplishing the same result and does not require you to declare additional private fields.

The constructor of `TelemetryEventArgs`, shown in Listing E-6, is used to set the `Temperature` property. I utilize this constructor in the `Telemetry` class to easily wrap the emulated temperature into an instance of the `TelemetryEventArgs`.

LISTING E-6 TelemetryEventArgs constructor

```
TelemetryEventArgs::TelemetryEventArgs(double temperature)
{
    Temperature = temperature;
}
```

Concurrency

To implement the background operation, I create the `Telemetry` class. As shown in Listing E-7, this class declares the `DataReady` event and the `OnDataReady` inline function that raises the event. This additional function is required because in C++/CX, you cannot directly raise an event in the class definition without writing custom raise logic (see *https://bit.ly/cpp_cx_events*).

LISTING E-7 Declaration of the Telemetry class

```
ref class Telemetry sealed
{
public:
    event DataReadyEventHandler^ DataReady;

    void OnDataReady(Object^ sender, TelemetryEventArgs^ e)
    {
        DataReady(this, e);
    }

    property bool IsActive
    {
        public: bool get() { return isActive; }
        private: void set(bool value) { isActive = value; }
    }

    void Start();
    void Stop();

private:
    const int msDelayTime = 1000;
    bool isActive;

    task<void> telemetryTask;
    cancellation_token_source *telemetryCancellationTokenSource;

    void InitializeTelemetryTask();
    TelemetryEventArgs^ GetSensorReading();
};
```

I declare the `IsActive` property, which is the flag indicating whether the background operation is in progress. I also have two public methods—`Start` and `Stop`—which are used to control the thread, in which I periodically emulate sensor readings at the delays specified by the `msDelayTime` constant.

Going further, the `Telemetry` class declares two private fields: `telemetryTask` and `telemetryCancellationTokenSource`. They are of types `task<T>` and `cancellation_token_source`, respectively. Both types are parts of the Parallel Patterns Library (PPL; *https://bit.ly/pp_lib*). PPL provides the API that you can use to run concurrent operations within the Windows ThreadPool. Additionally, PPL implements generic concurrent algorithms (like parallel loops) and parallel containers.

I instantiate `telemetryTask` and `telemetryCancellationTokenSource` within `InitializeTelemetryTask`, shown in Listing E-8. First, I create `cancellation_token_source`, which is used to cancel the task. Note that `cancellation_token_source` is not a managed type, and thus is declared as the pointer type and created with the `new` keyword. I instantiate the task class with void as a generic argument. The lambda used there retrieves the current class instance in the capture clause (`[this]`). The task method itself looks similar to the C# examples. Namely, I generate a sensor reading, and then report it to the listeners using an event. These operations are delayed in time using the `Sleep` function.

LISTING E-8 Telemetry task initialization

```
void Telemetry::InitializeTelemetryTask()
{
    telemetryCancellationTokenSource = new cancellation_token_source();

    telemetryTask = task<void>([this]
    {
        auto cancellationToken = telemetryCancellationTokenSource->get_token();

        while (!cancellationToken.is_canceled())
        {
            if (IsActive)
            {
                auto telemetryEventArgs = GetSensorReading();

                OnDataReady(this, telemetryEventArgs);

                Sleep(msDelayTime);
            }
        }

    }, telemetryCancellationTokenSource->get_token());
}
```

To emulate sensor readings, I write the `GetSensorReading` method, which appears in Listing E-9. To emulate temperature changes, `GetSensorReading` uses randomly generated values, which are added to the base temperature of 35 degrees Celsius.

LISTING E-9 Emulating temperature reading

```
TelemetryEventArgs^ Telemetry::GetSensorReading()
{
    const double baseTemp = 35.0;
    const double interval = 2.5;

    auto temp = baseTemp + interval * rand() / double(RAND_MAX);

    return ref new TelemetryEventArgs(temp);
}
```

The telemetry is started by invoking the `Start` method, shown in Listing E-10. Note that, after initializing the task, I do not need to invoke any additional methods that explicitly run the concurrent operation. The task method from Listing E-8 is automatically queued on the ThreadPool and started. Therefore, I set the `IsActive` flag to `true` before invoking `InitializeTelemetryTask`.

LISTING E-10 Starting telemetry as the background operation

```
void Telemetry::Start()
{
    if (!IsActive)
    {
        IsActive = true;

        InitializeTelemetryTask();
    }
}
```

The background operation is stopped using the method from Listing E-11. It sends the signal to the task using `cancellation_token_source` and sets `IsActive` flag to `false`. I also needed to manually release resources used by `cancellation_token_source`, because it was a pointer variable.

LISTING E-11 Telemetry is stopped by sending a cancellation signal to the task

```
void Telemetry::Stop()
{
    if (IsActive)
    {
        telemetryCancellationTokenSource->cancel();

        IsActive = false;

        delete telemetryCancellationTokenSource;
        telemetryCancellationTokenSource = nullptr;
    }
}
```

Data binding

The last element of the SenseHat.Sensors.CppCx app that requires additional discussion is the way you bind properties of the view models to the UI. Again, the XAML declaration stays unchanged. However, there are some differences in the logic. The class implementing your view model has to be associated with `Windows::UI::Xaml::Data::BindableAttribute`.

Here, my view model is implemented within the `SensorsViewModel` class. (See the companion code at Appendix E/ViewModels.) Its declaration appears in Listing E-12. As in C#, the `SensorsViewModel` class implements the `INotifyPropertyChanged` interface. Hence, `SensorsViewModel` declares the event `PropertyChanged` of type `PropertyChangedEventHandler`. This event is raised within the `OnPropertyChanged` method. Basically, this looks the same as in C#. However, in C++/CX, the `CallerMemberNameAttribute` is unavailable, so you need to explicitly pass the property name to the `OnPropertyChanged` method. In particular, `OnPropertyChanged` is invoked in the setting of the `Temperature` and `IsTelemetryActive` properties. Whenever that happens, the data binding mechanism handles the `PropertyChanged` event to update the corresponding elements of the UI. The temperature value is displayed in the UI through the `Temperature` property, while `IsTelemetryActive` enables or disables Start Sensor Readings and Stop Sensor Readings buttons.

LISTING E-12 Declaration of the SensorsViewModel class

```
[Bindable]
public ref class SensorsViewModel sealed : INotifyPropertyChanged
{
public:
    SensorsViewModel();

    virtual event PropertyChangedEventHandler^ PropertyChanged;

    property double Temperature
    {
        double get() { return temperature; }
        void set(double value)
        {
            temperature = value;
            OnPropertyChanged("Temperature");
        }
    }

    property bool IsTelemetryActive
    {
        bool get() { return isTelemetryActive; }
        void set(bool value)
        {
            isTelemetryActive = value;
            OnPropertyChanged("IsTelemetryActive");

            ToggleStartStopButtons(!value);
        }
```

```
    }

    property bool IsStartSensorReadingButtonEnabled
    {
        bool get() { return isStartSensorReadingButtonEnabled; }
        private: void set(bool value) { isStartSensorReadingButtonEnabled = value; }
    }

    property bool IsStopSensorReadingButtonEnabled
    {
        bool get() { return isStopSensorReadingButtonEnabled; }
        private: void set(bool value) { isStopSensorReadingButtonEnabled = value; }
    }

    private:
        bool isTelemetryActive;
        bool isStartSensorReadingButtonEnabled;
        bool isStopSensorReadingButtonEnabled;

        double temperature;

        void ToggleStartStopButtons(bool isStartEnabled);
        void OnPropertyChanged(String^ propertyName);
        {
            PropertyChanged(this, ref new PropertyChangedEventArgs(propertyName));
        }
};
```

The definition of `SensorsViewModel` is shown in Listing E-13 and contains constructor and a `ToggleStartStopButtons` method. The latter sets `IsStartSensorReadingButtonEnabled` and `IsStopSensorReadingButtonEnabled` to reflect the current telemetry status in the UI. Hence, at startup, only the Start Sensor Readings button is enabled (see the constructor of `SensorsViewModel` in Listing E-13).

LISTING E-13 Definition of the SensorsViewModel class

```
SensorsViewModel::SensorsViewModel()
{
   ToggleStartStopButtons(true);
}

void SensorsViewModel::ToggleStartStopButtons(bool isStartEnabled)
{
   IsStartSensorReadingButtonEnabled = isStartEnabled;
   OnPropertyChanged("IsStartSensorReadingButtonEnabled");

   IsStopSensorReadingButtonEnabled = !isStartEnabled;
   OnPropertyChanged("IsStopSensorReadingButtonEnabled");
}
```

Value converters

As in C#, you can also use value converters, which transform data transmitted through the data binding. Here, I implement such a converter to format temperature displayed in the UI. The declaration of `TemperatureToStringConverter`, which appears in Listing E-14, shows that the value converter class has to implement the `IValueConverter` interface. Hence, `TemperatureToStringConverter` has two methods: `Convert` and `ConvertBack`. Their meaning is the same as in previous C# samples.

LISTING E-14 Converter declaration

```
public ref class TemperatureToStringConverter sealed : IValueConverter
{
public:
    virtual Object ^Convert(Object ^value, TypeName targetType, Object ^parameter,
        String ^language);
    virtual Object ^ConvertBack(Object ^value, TypeName targetType, Object ^parameter,
        String ^language);
};
```

To perform an actual conversion and prepare the temperature string, I use standard C/C++ techniques (see Listing E-15). First, I convert `Object^` to `double` using the `static_cast` template construct. Then, I format the temperature and supplement it with a °C with the `sprintf_s` function. The result of this operation is written to the array of `wchar_t` (wide character type). Finally, I convert this array to `String^` using a dedicated constructor of that class. This is another example proving how easily you can mix managed code with native code. You do pay a price in terms of time required to write the app logic (compared to C#).

LISTING E-15 Converter definition

```
Object ^TemperatureToStringConverter::Convert(Object ^value, TypeName targetType,
    Object ^parameter, String ^language)
{
    String^ result = "Unavailable";

    try
    {
        auto temperature = static_cast<double>(value);

        wchar_t buffer[100];
        wchar_t degChar = (wchar_t)176;

        swprintf_s(buffer, L"%.2f %cC", temperature, degChar);

        result = ref new String(buffer);
    }
    catch (Exception^) {}

    return result;
}
```

```
Object ^TemperatureToStringConverter::ConvertBack(Object ^value, TypeName targetType,
    Object ^parameter, String ^language)
{
    throw ref new NotImplementedException();
}
```

To use the converter, you need to declare it in the control, page, or app resources. Here, I declare `TemperatureToStringConverter` in the app resources, so I modify the App.xaml file as shown in Listing E-16. To make `TemperatureToStringConverter` "visible" in the app resources, I have to include the appropriate header file in App.xaml.h:

```
#include "TemperatureToStringConverter.h"
```

LISTING E-16 Converter declaration in the application resources

```
<Application
    x:Class="SenseHat_Sensors_CppCx.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:converters="using:SenseHat.Sensors.Converters"
    RequestedTheme="Light">

    <Application.Resources>
        <converters:TemperatureToStringConverter x:Key="TemperatureToStringConverter" />
    </Application.Resources>
</Application>
```

Summary

In this appendix, I described features of C++/CX that you may find useful for implementing UWP IoT apps with C++/CX. The functionality developed here was dedicated to periodically reading data from sensors. Although I used emulated readings, you can quite easily replace the emulated values with the real data obtained from the Sense HAT add-on board.