APPENDIX A

비주얼 베이직과 자바스크립트를 사용한 LED 제어 코드 예제

부록 A에서는 2장, '디바이스용 유니버설 Windows 플랫폼'에서 개발한 LED 회로 제어의 추가 예제를 살펴본다. 다음 샘플 애플리케이션은 UWP를 액세스하는 다른 방법을 소개하며, 추가 기능은 보이지 않는다.

비주얼 베이직/XAML을 사용한 UI 있는 앱

다음의 절차를 따라서 IoT Hello, World! 앱을 구현한다.

1. **새 프로젝트 만들기** 대화 상자를 실행한다.
2. **새 프로젝트 만들기** 대화 상자에서 다음의 절차를 수행한다.
   a. 검색 상자에서 **Visual Basic**를 입력한다.
   b. **비어 있는 앱(유니버설 Windows)** 프로젝트 템플릿을 선택한 뒤 **다음**을 클릭한다.
   c. 프로젝트 이름을 **HelloWorldIoTVB**로 바꾸고 적절한 위치를 선택한 뒤 **만들기**를 클릭한다.
3. **새 유니버설 Windows 플랫폼 프로젝트** 대화 상자에서 **확인**을 클릭한다.
4. MainPage.xaml.vb를 열고 예제 A-1의 내용으로 수정한다.
5. 대상 플랫폼을 **ARM**으로 변경하고 원격 컴퓨터로 IoT 디바이스를 지정한다(3장의 'C#/XAML' 절에서 절차 참고).

6. 디버깅을 시작한다. 애플리케이션이 IoT 디바이스에 배포되고 실행된다.

예제 A-1 비주얼 베이직으로 LED 제어 구현

```
Imports Windows.Devices.Gpio

Public NotInheritable Class MainPage
    Inherits Page

    Private Const gpioPinNumber = 5
    Private Const msShineDuration = 4000

    Protected Overrides Sub OnNavigatedTo(e As NavigationEventArgs)
        MyBase.OnNavigatedTo(e)

        BlinkLed(gpioPinNumber, msShineDuration)
    End Sub

    Private Function ConfigureGpioPin(pinNumber As Integer) As GpioPin
        Dim gpioControl = GpioController.GetDefault()
        Dim pin As GpioPin = Nothing

        If gpioControl IsNot Nothing Then
            pin = gpioControl.OpenPin(pinNumber)

            If pin IsNot Nothing Then
                pin.SetDriveMode(GpioPinDriveMode.Output)
            End If
        End If

        Return pin
    End Function

    Private Sub BlinkLed(gpioPinNumber As Integer, msShineDuration As Integer)
        Dim pin = ConfigureGpioPin(gpioPinNumber)

        If pin IsNot Nothing Then
            pin.Write(GpioPinValue.Low)

            Task.Delay(msShineDuration).Wait()

            pin.Write(GpioPinValue.High)
        End If
    End Sub
End Class
```

비주얼 베이직 UWP 프로젝트의 구조는 C# 프로젝트와 아주 비슷하다. 즉 `MainPage` 뷰는 MainPage.xaml와 MainPage.xaml.vb라는 2개의 파일 내에서 구현된다. 후자는 4초간 LED를 켜는 로직을 구현한다. 애플리케이션의 일반적인 흐름은 C#과 C++ 프로젝트의 경우와 같다. 즉 먼저 `GpioController` 클래스의 인스턴스에 대한 참조를 얻은 다음 GPIO 포트를 여는 데 사용한다. LED 회로가 액티브-로active-low 상태로 배치됐다고 가정하면 GPIO 포트는 로low 상태로 설정해 LED에 전원을 인가하고, 하이 상태로 설정해 다이오드를 끈다.

비주얼 베이직/XAML을 사용한 UI 없는 앱

다음 절차를 따라서 비주얼 베이직을 사용해 백그라운드 IoT 애플리케이션을 구현한다.

1. **Background Application (IoT)** 프로젝트 템플릿을 사용해 새로운 IoTBackground AppVB 앱을 만든다. **새 프로젝트 만들기** 대화 상자의 검색 상자에서 **Background IoT**를 입력해 이 템플릿을 찾을 수 있다.
2. **Windows IoT Extensions for the UWP**를 참조한다.
3. StartupTask.vb 파일을 예제 A-2의 내용으로 수정한다.
4. 앱을 IoT에 배포한다. LED가 깜박이기 시작한다.

예제 A-2 The contents of a StartupTask.vb file

```
Imports Windows.ApplicationModel.Background
Imports Windows.Devices.Gpio

Public NotInheritable Class StartupTask
    Implements IBackgroundTask

    Private Const gpioPinNumber = 5
    Private Const msShineDuration = 4000

    Public Sub Run(taskInstance As IBackgroundTaskInstance) Implements
IBackgroundTask.Run
```

```
        BlinkLed(gpioPinNumber, msShineDuration)
    End Sub

    Private Function ConfigureGpioPin(pinNumber As Integer) As GpioPin
        Dim gpioControl = GpioController.GetDefault()
        Dim pin As GpioPin = Nothing

        If gpioControl IsNot Nothing Then
            pin = gpioControl.OpenPin(pinNumber)

            If pin IsNot Nothing Then
                pin.SetDriveMode(GpioPinDriveMode.Output)
            End If
        End If

        Return pin
    End Function

    Private Sub BlinkLed(gpioPinNumber As Integer, msShineDuration As Integer)
        Dim ledGpioPin = ConfigureGpioPin(gpioPinNumber)

        If ledGpioPin IsNot Nothing Then
            While True
                SwitchGpioPin(ledGpioPin)
                Task.Delay(msShineDuration).Wait()
            End While
        End If
    End Sub

    Private Sub SwitchGpioPin(gpioPin As GpioPin)
        Dim currentPinValue = gpioPin.Read()
        Dim newPinValue = InvertGpioPinValue(currentPinValue)

        gpioPin.Write(newPinValue)
    End Sub

    Private Function InvertGpioPinValue(currentPinValue As GpioPinValue) As
GpioPinValue
        Dim invertedGpioPinValue As GpioPinValue

        If currentPinValue = GpioPinValue.High Then
            invertedGpioPinValue = GpioPinValue.Low
        Else
            invertedGpioPinValue = GpioPinValue.High
```

```
        End If

        Return invertedGpioPinValue
    End Function
End Class
```

HTML, CSS, 자바스크립트를 사용한 UI 있는 앱

이 절의 내용은 비주얼 스튜디오 2017에서만 지원한다.

다음의 절차를 따라서 UI 있는 앱을 구현한다.

1. **새 프로젝트 만들기** 대화 상자를 실행한다.
2. 검색 상자에 JavaScript를 입력한 뒤 프로젝트 템플릿 목록에서 WinJS App(Universal Windows)을 선택한다.
3. 프로젝트 이름을 HelloWorldIoTJS으로 변경하고 **확인** 버튼을 클릭한다.
4. Windows IoT Extensions for the UWP를 참조한다.
5. 예제 A-3의 내용으로 main.js 파일을 수정한다.
6. 대상 플랫폼을 ARM으로 변경한다.
7. 프로젝트 속성 대화 상자를 열고 다음을 수행한다.
    a. 플랫폼 설정을 ARM으로 변경한다.
    b. **시작할 디버거** 드롭다운 목록에서 **원격 컴퓨터**를 선택한다.
    c. **컴퓨터 이름** 입력 상자에서 〈찾기 ...〉를 사용해 IoT 디바이스를 찾는다(그림 A-1 참고).
8. 앱 디버깅을 시작한다.

예제 A-3 자바스크립트를 사용한 GPIO 포트 제어

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    var gpio = Windows.Devices.Gpio;
    var gpioPinNumber = 5;
    var msShineDuration = 1000;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !==
                activation.ApplicationExecutionState.terminated) {
            } else {
            }
            args.setPromise(WinJS.UI.processAll());

            blinkLed(gpioPinNumber, msShineDuration);
        }
    };

    function configureGpioPin(pinNumber) {
        var gpioController = gpio.GpioController.getDefault();

        var gpioPin = null;

        if (gpioController) {
            gpioPin = gpioController.openPin(pinNumber);

            if (gpioPin) {
                gpioPin.setDriveMode(gpio.GpioPinDriveMode.output);
            }
        }

        return gpioPin;
    }

    function blinkLed(pinNumber, msShineDuration) {
        var ledGpioPin = configureGpioPin(pinNumber);

        if (ledGpioPin) {
            ledGpioPin.write(gpio.GpioPinValue.low);
```

```
            setTimeout(function () {
                ledGpioPin.write(gpio.GpioPinValue.high);
            }, msShineDuration);
        }
    }

    app.oncheckpoint = function (args) {
    };

    app.start();
})();
```

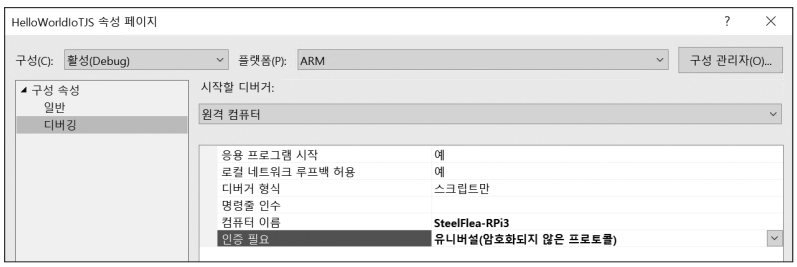

그림 A-1 자바스크립트 유니버설 Windows 앱 디버깅 구성

비주얼 베이직 샘플의 경우처럼 이 애플리케이션은 IoT 디바이스로 배포돼 실행된다. 잠시 뒤에 적절한 LED가 1초 동안 켜진다.

WinJS App(Universal Windows)을 사용해 생성된 프로젝트 골격을 살펴보자. 애플리케이션 매니페스트 파일과 함께 다음 요소를 포함한다.

- **CSS 폴더**: 스타일 시트 저장소
- **images 폴더**: 이미지 리소스에 대한 템플릿 제공 위치
- **js 폴더**: 자바스크립트 파일 저장소
- **lib 폴더**: HTML/CSS UWP 애플리케이션을 만들기 위한 자바스크립트 라이브러리 파일 저장소
- **index.html**: 애플리케이션 메인 뷰(페이지)의 UI 선언

기본적으로 CSS와 js 폴더에는 각각 default.css와 main.js라는 파일이 있다. 이들 파일은 index.html과 함께 자바스크립트 UWP 애플리케이션의 메인 페이지를 구현한다.

IoTBackgroundAppJS의 UI 선언을 수정하지는 않지만, 뷰의 로직을 구현하는 main.js 자바스크립트 파일의 기본 내용을 업데이트한다. 이 파일은 UWP 애플리케이션을 시작하고 이벤트 핸들러를 애플리케이션 onactivated 이벤트 핸들러에 연결하는 단일 비동기 함수로 이뤄져 있다. 이 핸들러 내에서 앞서처럼(액티브-로 상태로 가정) GPIO 핀을 구성하고 출력-로output-low 상태로 구동하는 blinkLed 메서드를 호출한다. 이어서 GPIO 핀을 출력-하이output-high 상태로 구동해 LED를 오프한다. gpioPin 개체가 노출한 write 메서드의 호출 사이의 지연을 구현하고자 setTimeout 자바스크립트 함수를 사용한다.

자바스크립트 앱의 진입점

비주얼 베이직 앱의 진입점은 C#과 C++ 프로젝트와 유사하기 때문에 따로 설명하지 않았다. 하지만 자바스크립트/HTML UWP 애플리케이션은 XAML 애플리케이션과 약간 다른 방식으로 활성화된다. 여기에는 Main 메서드를 구현하는 정적 Program 클래스가 없다. 대신 WWAHost.exe 프로세스의 인스턴스인 호스팅 환경이 package.appxmanifest 파일에서 가리키는 시작 페이지를 로드한다. HelloWorldIoTJS 프로젝트의 경우 이 파일은 index.html 파일 내에 정의된 페이지를 가리킨다. 이 구성은 예제 A-4에서 보인 매니페스트 파일에서 적절한 항목을 직접 편집하거나 Visual Studio package.appxmanifest 편집기(그림 A.2 참고)를 사용해 바꿀 수 있다.

예제 A-4 HelloWorldIoTJS 앱 매니페스트 파일의 일부

```
<Applications>
    <Application
      Id="App"
      StartPage="index.html">
      <uap:VisualElements
        DisplayName="HelloWorldIoTJS"
        Description="HelloWorldIoTJS"
```

```
        BackgroundColor="transparent"
        Square150x150Logo="images\Square150x150Logo.png"
        Square44x44Logo="images\Square44x44Logo.png">

        <uap:DefaultTile Wide310x150Logo="images\Wide310x150Logo.png" />
        <uap:SplashScreen Image="images\splashscreen.png" />

      </uap:VisualElements>
    </Application>
</Applications>
```

그림 A-2 HelloWorldIoTJS 애플리케이션의 package.appxmanifest 편집기. 기본 애플리케이션 뷰를 구현하는 파일의 이름을 강조한다.

자동으로 생성된 index.html 파일을 예제 A-5에서 나타냈다. 이 파일은 전형적인 HTML 마크업을 포함하며, 앱이 시작될 때 WWAHost.exe가 로드해 처리한다. WWAHost.exe가 자바스크립트 파일 참조를 만날 경우(WinJS 라이브러리와 예제 A-6의 main.js의 앱 시작 코드), 그 코드를 로드한 다음 실행한다. main.js는 앱 초기화를 위한 자기 실행 익명 함수 self-executing anonymous function로 만들어졌으며, 앱 시작과 `Application` 개체를 검색해 앱이 시작되고 일시 중지될 때 각각 실행되는 `onactivated`와 `oncheckpoint` 이벤트에 대한 리스너를 설정한다.

예제 A-5 HelloWorldIoTJS 프로젝트의 index.html 파일 내용

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>HelloWorldIoTJS</title>
    <link href="lib/winjs-4.0.1/css/ui-light.css" rel="stylesheet" />
    <script src="lib/winjs-4.0.1/js/base.js"></script>
    <script src="lib/winjs-4.0.1/js/ui.js"></script>
    <link href="css/default.css" rel="stylesheet" />
    <script src="js/main.js"></script>
</head>
<body class="win-type-body">
    <div>Content goes here!</div>
</body>
</html>
```

예제 A-6 기본 JS UWP 앱 진입점

```
(function () {
    "use strict";

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !==
                activation.ApplicationExecutionState.terminated) {
            } else {
            }
            args.setPromise(WinJS.UI.processAll());
        }
    };

    app.oncheckpoint = function (args) {
    };

    app.start();
})();
```

다음 단계를 따라서 UI 있는 자바스크립트 앱을 구현한다(비주얼 스튜디오 2015에서만 사용 가능).

1. **새 프로젝트** 대화 상자를 실행한다.
2. 검색 상자에 JavaScript를 입력한 뒤 Background Application (IoT) 프로젝트 템플릿을 선택한다.
3. 프로젝트 이름을 IoTBackgroundAppJS로 변경하고 **확인** 버튼을 클릭한다.
4. Windows IoT Extensions for the UWP를 참조한다.
5. 예제 A-7의 내용으로 startuptask.js를 수정한다.
6. IoT 디바이스에 앱을 배포한다. 잠시 뒤 LED가 깜박이기 시작한다.

예제 A-7 IoT 백그라운드 애플리케이션의 자바스크립트 구현

```
(function () {
    "use strict";

    var gpio = Windows.Devices.Gpio;
    var gpioPinNumber = 5;
    var msShineDuration = 1000;

    function configureGpioPin(pinNumber) {
        var gpioController = gpio.GpioController.getDefault();

        var gpioPin = null;

        if (gpioController) {
            gpioPin = gpioController.openPin(pinNumber);

            if (gpioPin) {
                gpioPin.setDriveMode(gpio.GpioPinDriveMode.output);
            }
        }

        return gpioPin;
    }

    function switchGpioPin(gpioPin) {
        var currentPinValue = gpioPin.read();
```

```
        var newPinValue = !currentPinValue;

        gpioPin.write(newPinValue);
    }

    function blinkLED(pinNumber, msShineDuration) {
        var ledGpioPin = configureGpioPin(pinNumber);

        if (ledGpioPin) {
            setInterval(function () {
                switchGpioPin(ledGpioPin);
            }, msShineDuration);
        }
    }

    blinkLED(gpioPinNumber, msShineDuration);
})();
```

APPENDIX B

라즈베리 파이 2 HDMI 모드

부록 B는 라즈베리 파이 2 IoT 디바이스의 HDMI 디스플레이 구성으로 사용할 수 있는 화면 해상도 목록이다. 표 B-1은 hdmi_group=1일 경우 유효한 모드를 나타냈으며, 표 B-2는 hdmi_group=2일 경우의 해상도 모드를 나타냈다.

표 B-1 hdmi_group=1일 경우 HDMI 모드 설정

HDMI 모드 설정	해상도	재생률(Hz)
1	VGA	60
2	480p	60
3	480p	60
4	720p	60
5	1080i	60
6	480i	60
7	480i	60
8	240p	60
9	240p	60
10	480i	60
11	480i	60
12	240p	60
13	240p	60
14	480p	60
15	480p	60
16	1080p	50
17	576p	50
18	576p	50

HDMI 모드 설정	해상도	재생률(Hz)
19	720p	50
20	1080i	50
21	576i	50
22	576i	50
23	288p	50
24	288p	50
25	576i	50
26	576i	50
27	288p	50
28	288p	50
29	576p	50
30	576p	50
31	1080p	24
32	1080p	25
33	1080p	30
34	1080p	60
35	480p	60
36	480p	50

HDMI 모드 설정	해상도	재생률(Hz)
37	576p	50
38	576p	50
39	1080i	100
40	1080i	100
41	720p	100
42	576p	100
43	576p	100
44	576i	100
45	576i	120
46	1080i	120
47	720p	120
48	480p	120

HDMI 모드 설정	해상도	재생률(Hz)
49	480p	120
50	480i	120
51	480i	200
52	576p	200
53	576p	200
54	576i	200
55	576i	240
56	480p	240
57	480p	240
58	480i	240
59	480i	60

표 B-2 hdmi_group=2 경우 HDMI 모드 설정

HDMI 모드 설정	해상도	재생률(Hz)
1	640×350	85
2	640×400	85
3	720×400	85
4	640×480	60
5	640×480	72
6	640×480	75
7	640×480	85
8	800×600	56
9	800×600	60
10	800×600	72
11	800×600	75
12	800×600	85
13	800×600	120
14	848×480	60
15	1024×768	43
16	1024×768	60

HDMI 모드 설정	해상도	재생률(Hz)
17	1024×768	70
18	1024×768	75
19	1024×768	85
20	1024×768	120
21	1152×864	75
22	1280×768	60 (깜박임 감소)
23	1280×768	60
24	1280×768	75
25	1280×768	85
26	1280×768	120
27	1208×800	60 (깜박임 감소)
28	1208×800	60
29	1208×800	75
30	1208×800	85
31	1208×800	120
32	1208×960	60

HDMI 모드 설정	해상도	재생률(Hz)
33	1208×960	85
34	1208×960	120
35	1208×1024	60
36	1208×1024	75
37	1208×1024	85
38	1208×1024	120
39	1360×768	60
40	1360×768	120
41	1400×1050	60 (깜박임 감소)
42	1400×1050	60
43	1400×1050	75
44	1400×1050	85
45	1400×1050	120
46	1440×900	60 (깜박임 감소)
47	1440×900	60
48	1440×900	75
49	1440×900	85
50	1440×900	120
51	1600×1200	60
52	1600×1200	65
53	1600×1200	70
54	1600×1200	75
55	1600×1200	85
56	1600×1200	120
57	1680×1050	60 (깜박임 감소)
58	1680×1050	60
59	1680×1050	75

HDMI 모드 설정	해상도	재생률(Hz)
60	1680×1050	85
61	1680×1050	120
62	1792×1344	60
63	1792×1344	75
64	1792×1344	120
65	1856×1392	60
66	1856×1392	75
67	1856×1392	120
68	1920×1200	60 (깜박임 감소)
69	1920×1200	60
70	1920×1200	75
71	1920×1200	85
72	1920×1200	120
73	1920×1440	60
74	1920×1440	75
75	1920×1440	120
76	2560×1600	60 (깜박임 감소)
77	2560×1600	60
78	2560×1600	75
79	2560×1600	85
80	2560×1600	120
81	1366×768	60
82	1080p	60
83	1600×900	깜박임 감소
84	2048×1152	깜박임 감소
85	720p	60
86	1366×768	깜박임 감소

노트 | '깜박임 감소(reduced blanking)'는 연속적인 이미지들 사이의 휴지 간격이 줄어든 재생률(refresh rates)을 뜻한다.

APPENDIX C

비트, 바이트, 데이터 유형

부록 C에서는 바이트의 기본 표현을 보여 주고 데이터 유형을 구성하는 방법을 간략하게 설명한다. 또한 IoT 개발에 유용할 수 있는 C# 7.0의 몇 가지 새로운 기능을 보여 준다.

이진 인코딩: 정수 유형

바이트는 비트 배열, 즉 논리 값 배열로 해석될 수 있다. 의례적으로 이 배열의 각 항목은 바이트 값 V를 가중치 합계로 인코딩한다.

$$V = \sum_{i=0}^{N-1} b_i 2^i$$

여기서 b_i는 비트 값(0 또는 1), i는 비트 인덱스, N은 비트 수를 나타낸다. 2의 기수radix는 단일 비트가 2개의 값을 가질 수 있다는 사실에서 비롯된다. 합계 가중치는 비트 인덱스와 함께 증가한다. 이러한 이유로 인덱스 0의 비트는 가장 큰 인덱스를 차지하는 최상위 비트MSB, Most Significant Bit와는 달리 최하위 비트LSB, Least Significant Bit라고 한다. 표 C-1은 39, 127, 168, 255바이트의 비트 표현(N=8) 사례 몇 가지를 보여 준다.

표 C-1 정수의 이진 표현: 39, 127, 168, 255

	MSB							LSB	V
인덱스	7	6	5	4	3	2	1	0	–
논리 값	0	0	1	0	0	1	1	1	–
수치 값	0	0	32	0	0	4	2	1	39
논리 값	0	1	1	1	1	1	1	1	–
수치 값	0	64	32	16	8	4	2	1	127
논리 값	1	0	1	0	1	0	0	0	–
수치 값	128	0	32	0	8	0	0	0	168
논리 값	1	1	1	1	1	1	1	1	–
수치 값	128	64	32	16	8	4	2	1	255

표 C-1에서 볼 수 있듯이 V는 255(2^8-1)보다 크거나 0보다 작을 수 없다. 따라서 바이트는 양수(부호 없는) 값만 인코딩할 수 있다. 음수 값은 어떻게 처리해야 할까? 가장 널리 사용되는 방법은 다음과 같다.

- **부호화 절대치**signed magnitude **표현**: 1개의 비트(일반적으로 MSB)에 부호 정보를 저장한다. 그러나 이 방법에는 0 값을 나타내는 두 가지 방법이 포함돼 있다.
- **1의 보수**: 양수의 비트를 반전(1에서 0으로 또는 그 반대로 변경)하면 음수 값이 생성된다. 이 솔루션에는 캐리carry 플래그가 필요하다.
- **2의 보수**: 앞의 문제는 2의 보수법에서 비트를 반전하고, 결과 숫자에 1을 더해서 음의 값을 생성함으로써 해결할 수 있다. 즉 최상위 비트로 인코딩된 값에 -1을 곱하고, 다른 비트를 사용해 인코딩된 값의 합계에 더한다.

$$V_s = \sum_{i=0}^{N-1} b_i 2^i - b_{N-1} 2^{N-1}$$

따라서 단일 바이트는 -128(-2^7)에서 127(2^7-1) 사이의 정수를 인코딩할 수 있다. 표 C-2는 2의 보수를 사용해 인코딩된 부호 있는 바이트의 몇 가지 예를 보여 준다.

표 C-2 부호 있는 정수의 이진 표현 : −128, −74, −39, 127

	MSB							LSB	V_s
인덱스	7	6	5	4	3	2	1	0	-
논리 값	1	0	0	0	0	0	0	0	-
수치 값	1	0	0	0	0	0	0	0	−128
논리 값	1	0	1	1	0	1	1	0	-
수치 값	−128	0	32	16	0	4	2	0	−74
논리 값	1	1	0	1	1	0	0	1	-
수치 값	−128	64	0	16	8	0	0	1	−39
논리 값	0	1	1	1	1	1	1	1	-
수치 값	0	64	32	16	8	4	2	1	127

더 큰 정숫값을 인코딩하고자 비트 배열은 N=16(`short` 및 `ushort` 데이터 유형), N=32(`uint` 및 `int`), N=64(`long` 및 `ulong`)로 확장된다. 이러한 데이터 유형은 표 C-3에 지정된 범위 값을 저장할 수 있다.

표 C-3 2의 보수 인코딩 시스템에 대한 정수 범위(부호 없는 정수의 최솟값은 0)

N	16	32	64
최솟값(부호 있음)	−32678	−2147483648	−9223372036854775808
최댓값(부호 있음)	32767	2147483647	9223372036854775807
최댓값(부호 없음)	65535	4294967295	18446744073709551615

물론 정수 데이터 유형의 정확한 범위를 기억할 필요는 없다. 특정 범위는 정수 데이터 유형의 `MinValue` 및 `MaxValue` 상수에서 얻을 수 있다. 이를 설명하고자 간단한 UWP 앱을 작성했다. 함께 제공되는 코드 **Appendix C/DataTypeRanges**를 참고하자. 해당 코드는 다음과 같이 구현돼 있다.

- 사용자 정의 컨트롤인 `DataTypeRangesControl`(DataTypeRanges/DataTypeRanges Control)을 생성한다. 이 컨트롤은 특정 형식이 처리할 수 있는 최솟값 및 최댓값과 함께 데이터 유형 레이블을 표시하는 세 가지 `TextBlock` 컨트롤로 구성된다.

- `RangesViewModel` 클래스를 작성한다(예제 C-1 참고). 이 클래스는 DataTypeRanges 앱의 메인 뷰에 대한 `ViewModel` 역할을 수행한다. `RangesViewModel`의 특정 필드는 UI에 바인딩되므로 속성을 수동으로 다시 작성할 필요가 없다.

예제 C-1 정수 데이터 유형의 가능한 최솟값 및 최댓값 검색

```
public class RangesViewModel
{
    // N=8
    public byte UInt8MinValue { get; set; } = byte.MinValue;
    public byte UInt8MaxValue { get; set; } = byte.MaxValue;
    public sbyte Int8MinValue { get; set; } = sbyte.MinValue;
    public sbyte Int8MaxValue { get; set; } = sbyte.MaxValue;

    // N=16
    public ushort UInt16MinValue { get; set; } = ushort.MinValue;
    public ushort UInt16MaxValue { get; set; } = ushort.MaxValue;
    public short Int16MinValue { get; set; } = short.MinValue;
    public short Int16MaxValue { get; set; } = short.MaxValue;

    // N=32
    public uint UInt32MinValue { get; set; } = uint.MinValue;
    public uint UInt32MaxValue { get; set; } = uint.MaxValue;
    public int Int32MinValue { get; set; } = int.MinValue;
    public int Int32MaxValue { get; set; } = int.MaxValue;

    // N=64
    public ulong UInt64MinValue { get; set; } = ulong.MinValue;
    public ulong UInt64MaxValue { get; set; } = ulong.MaxValue;
    public long Int64MinValue { get; set; } = long.MinValue;
    public long Int64MaxValue { get; set; } = long.MaxValue;
}
```

IoT 장치 또는 로컬 컴퓨터에서 DataTypeRanges 앱을 컴파일하고 실행하면 데이터 유형 범위가 그림 C-1과 같이 표시된다.

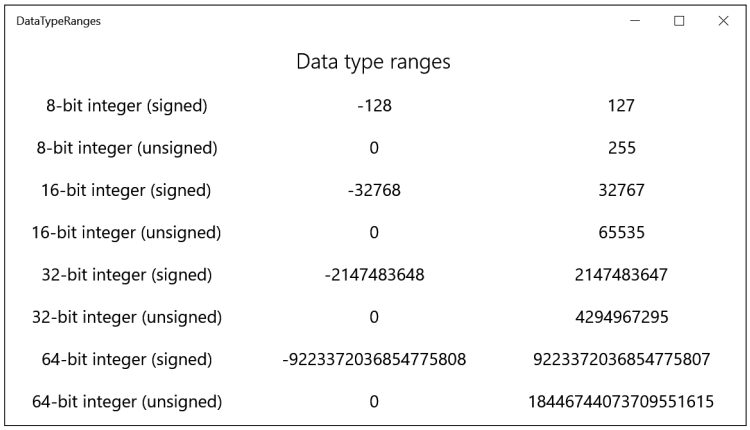

그림 C-1 `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` 데이터 유형의 상수 필드에서 검색된 데이터 유형 범위

이진 인코딩: 부동 소수점 수

일반적으로 부동 소수점 수는 다음과 같은 과학적 형식으로 표시된다.

$$V_f = s \times m \times 2^e$$

여기서 s는 부호, m은 가수(또는 유효값), e는 지수를 나타낸다. 마찬가지로 정수 데이터 유형에서와 같이 부호를 인코딩하는 데 단일 비트가 사용되는 반면 부호와 지수를 인코딩하는 데 특정 비트량이 사용된다. 지수는 고정 바이어스fixed bias가 추가된 부호 없는 8비트 정수로 저장된다. IEEE 754 표준에 따르면 이것은 32비트 정밀 부동 소수점 수($0 < m < 2^{24}$ 및 $-126 \leq e \leq 127$)에 해당한다. 즉 가수를 인코딩하는 데 23비트가 사용되는 반면 8비트는 바이어스 127(모두 0이 있는 바이트가 예약된다)로 지수를 인코딩한다.

64비트 정밀도 부동 소수점 수에서 52비트는 가수($0 < m < 2^{53}$)를 인코딩하고, 나머지 11비트는 1023의 바이어스로 $-1022 < e < 1023$ 범위의 값을 사용할 수 있는 지수를 인코딩한다.

이러한 개념을 처음 접하는 경우 DataTypeRange 앱을 확장해 float 및 double 데이터 유형의 최솟값과 최댓값을 표시하도록 하자.

16진수

IoT 프로그래밍에서는 레지스터 주소와 같은 값이 16진수(HEX)를 사용해 표시되는 경우가 매우 많다. 16진수는 10개의 숫자(0, 1, 2, …)와 6개의 알파벳(A=10, B=11, C=12, D=13, E=14, F=15)으로 구성된 16을 기수로 사용한다. 따라서 10진수(DEC) 정수는 16진수로 다음과 같이 나타낼 수 있다.

$$V_D = \sum_{i=0}^{N-1} h_i 16^i$$

여기서 h_i는 위치 i에서 사용 가능한 16진수 기호 중 하나다. 표 C-4에는 127, 255, 1024, 56506과 같은 부호 없는 정수의 10진수 및 16진수 표현이 포함돼 있다. LS 및 MS는 각각 최하위 및 최상위 요소를 나타낸다.

16진수 값 앞에는 0xFF와 같은 0x 기호가 표시된다.

표 C-4 정수의 16진수 및 10진수 표현: 127, 255, 1024, 56506

	MS			LS	VD
인덱스	3	2	1	0	-
16진수 값	0	0	7	F	-
10진수 값	0	0	112	15	127
16진수 값	0	0	F	F	-
10진수 값	0	0	240	15	255
16진수 값	0	4	0	0	-
10진수 값	0	1024	0	0	1024
16진수 값	D	C	B	A	-
10진수 값	53428	3072	176	10	56506

숫자 값 형식 지정

다행히도 다양한 숫자 체계 간의 변환을 수행하는 앱을 빠르게 작성할 수 있다. 이를 위해 숫자 체계 기수를 정의하는 적절한 숫자와 함께 정적 System.Convert.ToString 메서드를 사용한다. 2(2진수), 8(8진수), 10(10진수), 16(16진수) 값 중 하나를 사용할 수 있다. 예제 C-2는 정수를 2진수, 10진수, 16진수 형식으로 표현하는 방법을 보여 준다.

예제 C-2 다른 숫자 체계로 정숫값 표시

```
private static void ValueFormatting()
{
    var myInteger = 12345;

    Debug.Write("BIN: ");
    Debug.WriteLine(Convert.ToString(myInteger, 2));

    Debug.Write("DEC: ");
    Debug.WriteLine(Convert.ToString(myInteger, 10));

    Debug.Write("HEX: ");
    Debug.WriteLine(Convert.ToString(myInteger, 16));
}
```

위의 코드는 디버그 출력에서 다음 값을 생성한다.

```
BIN: 11000000111001
DEC: 12345
HEX: 3039
```

이진 리터럴 및 숫자 구분 기호

C# 7.0에는 저수준 프로그래밍에 유용한 두 가지 기능이 더 포함돼 있다. 이진 리터럴binary literals과 숫자 구분 기호digit separator다. 이진 리터럴을 사용하면 이진 상수를 정의할 수 있고, 숫자 구분 기호(_)를 사용하면 리터럴을 형식화할 수 있다. 이러한 기능을 활용하는 샘플 코드는 예제 C-3에 표시되고, 해당 출력은 예제 C-4에 표시된다.

예제 C-3 이진 리터럴

```
private static void BinaryLiterals()
{
    var a = 0b0001_1110;
    var b = 0b1000_0110;

    DebugValue(a);
    DebugValue(b);

    DebugValue(b & a);
    DebugValue(a | b);
    DebugValue(a ^ b);
}

private static void DebugValue(int value)
{
    Debug.Write(value.ToString().PadLeft(5));
    Debug.Write(", BIN: " + Convert.ToString(value, 2).PadLeft(8));
    Debug.WriteLine(", HEX: " + Convert.ToString(value, 16).PadLeft(2));
}
```

예제 C-4 예제 C-3의 BinaryLiterals 메서드의 샘플 출력

```
 30, BIN:    11110, HEX: 1e

134, BIN: 10000110, HEX: 86

  6, BIN:      110, HEX: 6

158, BIN: 10011110, HEX: 9e

152, BIN: 10011000, HEX: 98
```

위의 예제를 테스트하려면 비주얼 스튜디오 2019가 필요하다. 부록 F, 'IoT 개발을 위한 비주얼 스튜디오 2019 설정'에서는 C# 및 IoT 프로그래밍을 위한 개발 환경의 설정 방법을 설명한다.

APPENDIX D

Sense HAT 센서용 클래스 라이브러리

부록 D에서 이식 가능한 클래스 라이브러리PCL, Portable Class Library를 개발하고 다른 프로젝트에서 참조하는 방법을 설명한다. 이 개념은 여러 프로젝트에 걸쳐 공유할 수 있는 재사용 가능한 PCL을 만드는 것이다. 다른 플랫폼 간에 공통 C# 코드를 재사용할 수도 있다.

I^2C 인터페이스를 사용해 Sense HAT 센서 모두와 통신한다. 이 인터페이스를 통해 데이터를 읽고 쓰고자 UWP는 `I2cDevice` 클래스를 구현했다. 센서에서 원시 바이트를 받은 다음, 이들을 의미 있는 값으로 변환한다. 변환 로직은 UWP에 맞춰진 `I2cDevice` 개체와 달리 플랫폼 독립적이다. 따라서 변환 로직은 I^2C 통신을 처리하고자 `I2cDevice` 이외의 개체를 사용하는 다른 플랫폼에서 사용할 수 있다.

플랫폼 종속적인 코드에서 플랫폼 독립적인 부분을 분리하는 것이 좋다. 크로스 플랫폼 프로그래밍은 이런 접근 방식을 광범위하게 사용한다. 여기서는 Sense HAT 프로젝트(함께 제공되는 코드 Chapter 05/SenseHat)의 코드를 3개의 별도 프로젝트로 나눈다. 그림 D-1에서 보인 것처럼 계층적 구조의 솔루션이다.

- 맨 하위 계층은 플랫폼 독립적인 헬퍼와 변환 로직을 구현하는 PCL로 구성된다.
- PCL 위에는 UWP의 개체를 사용하는 변환기와 I^2C 관련 로직, 추가 헬퍼와 같은 UWP 클래스 라이브러리가 있다.
- 이 스택의 맨 위에는 PCL과 UWP 클래스 라이브러리의 코드를 사용하는 메인 UWP 앱이 있다.

함께 제공되는 Appendix D 폴더의 코드에서 전체 구현을 제공한다.

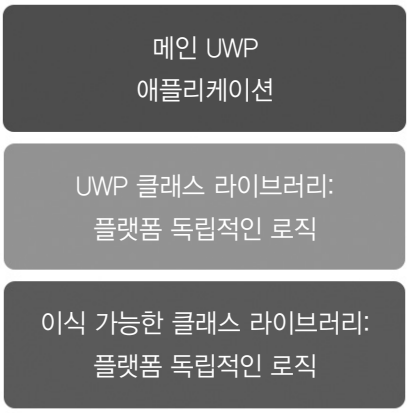

그림 D-1 플랫폼 독립적인 로직을 분리하고자 PCL을 활용하는 전형적인 프로젝트 구조

이식 가능한 클래스 라이브러리

다음의 절차를 따라서 PCL 프로젝트를 만든다.

1. **새 프로젝트 만들기** 대화상자를 실행한다.
2. 템플릿 검색 상자에서 **클래스 라이브러리**를 입력한다.
3. C#용 **클래스 라이브러리(.NET Standard)** 프로젝트를 선택한다.
4. 프로젝트 이름을 SenseHat.Portable로 설정하고, 솔루션 이름을 SenseHat으로 설정한다(그림 D-2 참고). 그다음 **만들기**를 클릭한다.

그림 D-2 SenseHat.Portable PCL을 만들기 위한 [새 프로젝트 구성] 대화 상자

SenseHat.Portable 프로젝트에 Helpers 폴더를 만든다(**SenseHat.Portable**에서 마우스 오른쪽 클릭한 다음, **추가**를 선택하고 **새 폴더** 클릭). 그다음 동일한 메뉴를 열고 **추가**를 선택한 다음, **기존 항목**을 선택해 SenseHat 프로젝트에서 다음 파일을 방금 만든 폴더에 추가한다(Chapter 05/SenseHat/Helpers에서 함께 제공하는 코드 참고).

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

- Vector3D.cs

마지막으로 모든 파일에 사용된 네임스페이스를 `SenseHat.Helpers`에서 `SenseHat.Portable.Helpers`로 변경한다.

유니버설 Windows 클래스 라이브러리

PCL이 준비되었다면 이제 UWP 클래스 라이브러리를 빌드하고자 다음 단계를 수행한다.

1. **솔루션 탐색기**에서 **SenseHat** 솔루션을 마우스 오른쪽 클릭하고, 콘텍스트 메뉴에서 **추가 > 새 프로젝트**를 선택한다.
2. **새 프로젝트 추가** 대화 상자의 **템플릿 검색** 상자에서 **클래스 라이브러리 유니버설**을 입력한다.
3. C#용 **클래스 라이브러리 (유니버설 Windows)** 프로젝트 템플릿을 선택하고, 이름을 **SenseHat.UWP**로 변경한 다음 **만들기** 버튼을 클릭한다.
4. **새 유니버설 Windows 플랫폼 프로젝트** 대화 상자에서 대상 버전과 최소 버전을 각각 **Windows 10, version 1903(10.0, 빌드 18362)**와 **Windows 10, version 1803(10.0, 빌드 17134)**로 설정한다.

이제 SenseHat.UWP 프로젝트에서 다음처럼 PCL을 참조한다.

1. **솔루션 탐색기**에서 SenseHat.UWP 프로젝트의 **참조** 노드를 오른쪽 클릭하고, 콘텍스트 메뉴에서 **참조 추가**를 선택한다. **참조 관리자** 대화 상자가 표시된다.
2. **참조 관리자** 대화 상자의 오른편 항목에서 **프로젝트/솔루션**을 찾아 **SenseHat.Portable**을 선택한다(그림 D-3 참고).

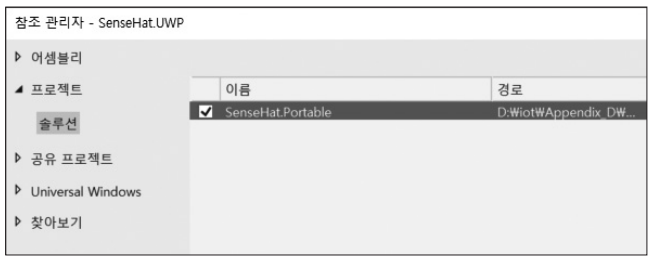

그림 D-3 SenseHat.UWP 프로젝트에서 SenseHat.Portable PCL 참조

3. 마지막으로 'Windows IoT Extensions for the UWP(버전 10.0.17134.0)'을 참조하고 **참조 관리자** 대화 상자를 닫는다.

SenseHat.UWP 클래스 라이브러리에서 Converters, Helpers, Sensors라는 3개의 폴더를 만들고, 5장, '센서의 데이터 판독'에서 보인 SenseHat 앱의 해당 폴더에 있는 적절한 파일을 추가한다. 따라서 SenseHat.UWP 프로젝트는 그림 D-4에서 보인 구조를 갖는다. SenseHat.UWP는 `I2cDevice` 클래스를 사용하는 모든 파일을 포함한다. 이 개체는 SenseHat.Portable로 타기팅한 플랫폼에서는 사용할 수 없다.

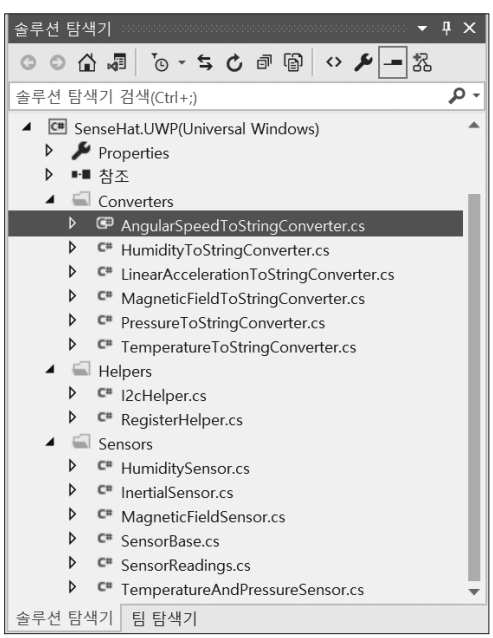

그림 D-4 SenseHat.UWP 프로젝트의 구조

또한 UWP 머리글자를 추가해 네임스페이스를 수정한다.

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

네임스페이스를 변경한 후 해당 `using` 문도 업데이트해야 한다. 이 작업을 비주얼 스튜디오 리팩터링 모듈을 사용해 자동으로 수행할 수 있다. 변수나 클래스, 구조체, 네임스페이스의 이름을 변경할 때마다 힌트(노란 불빛 전구)가 표시된다.

유니버설 Windows 플랫폼 애플리케이션

앞서의 클래스 라이브러리를 활용해 Sense HAT 애드온 보드에서 센서 판독 값을 표시하는 메인 애플리케이션을 만든다. 먼저 비주얼 C# '비어 있는 앱(유니버설 Windows)' 프로젝트 템플릿을 사용해 SenseHat.Sensors라는 새로운 프로젝트를 SenseHat 솔루션에 추가한다. 대상과 최소 버전을 SenseHat.UWP 클래스 라이브러리로 설정한다. 이어서 SenseHat.Portable과 SenseHat.UWP 프로젝트를 참조하고, SenseHat.Sensors를 시작 프로젝트로 설정한다(**SenseHat.Sensors**를 마우스 오른쪽 클릭하고 **시작 프로젝트로 설정**을 선택한다).

더 진행하기 전에 프로젝트 종속성과 빌드 순서를 검사한다. 전용 대화 상자를 사용해 이 종속성을 시각화할 수 있다. **솔루션 탐색기**에서 SenseHat 솔루션을 마우스 오른쪽 클릭하고, 콘텍스트 메뉴에서 **프로젝트 종속성**을 선택해 대화 상자를 실행한다. 그림 D-5에서 **프로젝트 종속성** 대화 상자 부분을 나타냈다. 이 대화 상자에는 **종속성**과 **빌드 순서**라는 두 가지 탭이 있다. 첫 번째 탭은 프로젝트의 계층 구조를 검토하고, 두 번째 탭은 솔루션을 빌드하는 방법을 확인한다.

그림 D-5 프로젝트 종속성과 SenseHat 솔루션의 빌드 순서

그림 D-5는 SenseHat.Portable 프로젝트가 먼저 컴파일된 다음, 컴파일러가 SenseHat.UWP 프로젝트를 빌드하고, 마지막으로 SenseHat.Sensors 프로젝트를 빌드하는 순서를 보여 준다. 이 순서는 SenseHat.Sensors 앱에서 사용하는 SenseHat.UWP가 SenseHat.Portable을 참조한다는 사실을 기반으로 한다.

SenseHat.Sensors 앱은 SenseHatTelemeter 앱과 동일한 방식으로 구현했다(12장, '원격 디바이스 모니터링' 참고) 즉 센서에서 주기적으로 데이터를 읽고자 `Telemetry`와 `TelemetryEvent-Args` 클래스를 구현했다. 두 가지 클래스는 압력과 선형, 각 가속도, 자기장 센서 판독을 포함하고자 `SenseHatTelemeter`에서 클래스를 확장해 만들었다.

`Telemetry` 클래스(Appendix D/SenseHat.Sensors/TelemetryControl/Telemetry.cs에서 함께 제공하는 코드 참고)는 public 생성자를 구현하지 않는다. 대신 예제 D-1에서 나타낸 정적 비동기 팩토리 메서드인 `CreateAsync`를 갖는다.

예제 D-1 Telemetry 클래스의 비동기 팩토리 메서드

```
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;
}
```

예제 D-1의 코드 조각을 살펴보면 여기서 센서를 초기화하기 때문에 `Telemetry` 클래스를 비동기 정적 팩토리 메서드를 사용해 만들어야 한다. 5장에서 보다시피 센서 클래스는 비동기 코드를 사용한다. 하지만 C#에서 생성자는 비동기가 될 수 없으므로 정적 비동기 팩토리 메서드를 사용한다.

private인 `InitializeSensors` 메서드를 예제 D-2에 나타냈다. 보다시피 센서를 나타내는 각 클래스의 `Initialize` 메서드를 호출한다. 그다음 `VerifyInitialization` 메서드(예제 D-2의 아래 부분)를 사용해 특정 클래스의 초기화 여부를 확인한다. 센서를 사용할 수 없다고 판단하면(초기화 되지 않음), 적절한 메시지로 예외를 던진다. 이 동작은 호출자에게 `Telemetry` 클래스가 모든 필요한 센서에 접근할 수 없으므로 더 이상 처리가 불가능함을 알려 준다.

예제 D-2 센서 초기화

```
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);
    }
}
```

성공적인 초기화 후 `Telemetry` 클래스 인스턴스의 `Start` 메서드를 사용해 주기적인 센서 판독 백그라운드 작업을 시작한다(예제 D-3 참고). 이 메서드는 먼저 원격 측정이 가능한 상태인지 검사한다. 사용 가능한 상태가 아니면 먼저 초기화한 다음 `Telemetry` 작업자 스레드를 시작한다.

예제 D-3 원격 측정 작업 시작하기

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

public void Start()
{
    if (!IsActive)
```

```
    {
        InitializeTelemetryTask();

        telemetryTask.Start();

        IsActive = true;
    }
}
```

Telemetry 작업자 스레드는 InitializeTelemetryTask 내에서 구성된다(예제 D-4 참고). 이 메서드는 센서 판독값을 주기적으로 가져와 DataReady 이벤트를 통해 리스너에 보고하고자 구현된 Task 인스턴스를 만든다. 이벤트를 일으키기 전에 예제 D-4의 while 루프 내에서 IsActive 플래그가 true인지도 검사한다.

예제 D-4 원격 측정 작업 초기화

```
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);
}
```

SenseHatTelemeter 앱에서처럼 전용 CancellationTokenSource를 통해 작업자 스레드(백그라운드 작업)를 중지할 수 있다(예제 D-5의 Stop 메서드).

예제 D-5 메서드 중지

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

        IsActive = false;
    }
}
```

작업자 스레드가 동작 중이면 센서 판독값은 예제 D-6의 GetSensorReadings 메서드를 사용해 가져올 수 있다. 이 함수는 temperatureAndPressureSensor.GetTemperature, inertialSensor.GetLinearAcceleration 등의 센서를 나타내는 클래스의 전용 메서드를 차례로 호출한다. 결과 값은 TelemetryEventArgs의 인스턴스로 래핑된 다음 리스너에 전달된다.

예제 D-6 센서에서 값 판독하기

```
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;
    }
}
```

그림 D-6은 앱이 데스크톱 플랫폼에서 실행될 때 SenseHat.Sensors의 UI를 나타냈다. 이 UI는 Control, Weather, Inertial이라는 3개의 탭으로 구성된다. 첫 번째 탭은 3개의 버튼이 있으며, 센서를 초기화하고 원격 측정을 시작하고 중지한다. Weather와 Inertial 탭은 센서의 값을 표시한다. Weather 탭은 온도, 습도, 압력을 표시하지만, Inertial 탭은 선형 가속도(가속도계), 각 속도(자이로스코프), 자기장(자력계)을 표시한다.

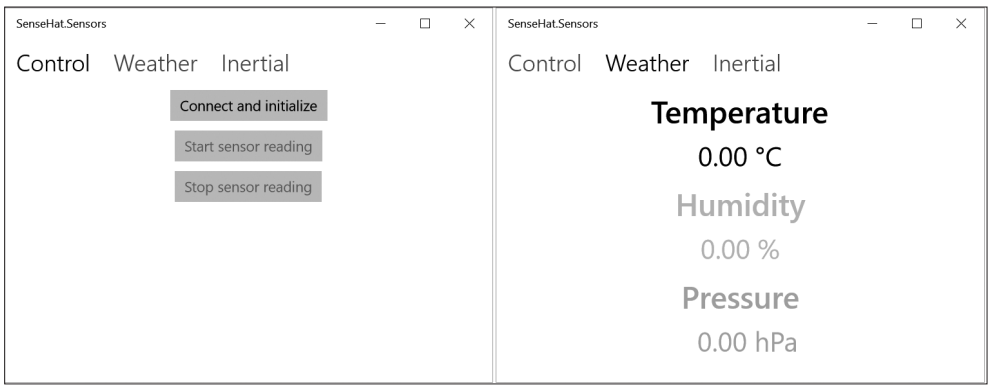

그림 D-6 데스크톱 플랫폼에서 실행되는 SenseHat.Sensors UI의 2개의 탭

Sense HAT 애드온 보드를 연결하고 센서 클래스를 초기화하고자 **Connect and Initialize** 버튼을 클릭한다. 그렇게 하면 예제 D-7의 `ButtonConnect_Click` 이벤트 핸들러를 호출한다. 이 메서드는 센서가 이미 초기화되었는지 검사한 다음 `Telemetry.CreateAsync` 메서드를 호출해 `Telemetry` 개체를 만든다. 그 뒤 핸들러는 새로운 센서 판독값을 사용할 수 있을 때마다 호출되는 `DataReady` 이벤트에 연결된다. 예제 D-7의 메서드는 조금 뒤에 설명하는 `SensorsViewModel` 클래스 인스턴스의 속성도 구성하고 있다.

예제 D-7 센서 초기화

```
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);
        }
    }
}
```

예제 D-8에서 보인 DataReady 이벤트의 이벤트 핸들러 내에서 TelemetryEventArgs의 인스턴스를 사용해 UI에서 센서 판독값을 표시한다. 이 작업은 SensorsViewModel 클래스의 해당 속성을 업데이트해 간접적으로 수행한다. DataReady 이벤트가 작업자 스레드에서 일어나므로 UI 스레드에 속성을 재작성하도록 넘겨야 한다.

예제 D-8 센서 판독값 표시하기

```
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;
    });
}
```

예제 D-9는 센서 판독을 시작하고 중지하는 이벤트 핸들러를 나타냈다. 이들 핸들러는 `Telemetry` 클래스 인스턴스의 해당 메서드를 간단히 호출하고 `SensorsViewModel` 속성을 통해 버튼 상태를 업데이트한다.

예제 D-9 원격 측정 시작 및 중지

```
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`은 상태를 제어하고 UI를 업데이트한다(Appendix D/SenseHat.Sensors/MainPage.xaml에서 함께 제공하는 코드 참고). `SensorsViewModel`의 주 요소는 다음의 여섯 가지 public 속성이다(Appendix D/SenseHat.Sensors/ViewModels/SensorsViewModel.cs에서 함께 제공하는 코드 참고).

- **SensorReadings**: 이 속성은 센서에서 얻은 값을 저장한다. `SensorReadings` 클래스의 멤버는 UI에 단방향으로 바인딩된다. `SensorsViewModel`의 `SensorReadings` 속성을 변경할 때마다 UI는 업데이트된다.
- **ReadoutDelay**: 이 속성은 연속 센서 판독 사이의 지연을 지정한다. 예제 D-7에서 구성한 기본값은 1초다.
- **IsConnected**: 이 속성은 `Telemetry` 클래스가 초기화되었는지의 여부를 가리킨다. 초기화되었다면 **Start Sensor Reading** 버튼이 활성화된다.
- **IsTelemetryActive**: 이 속성은 센서 판독의 백그라운드 작업이 실행 중인지를 UI에 알려준다. 실행 중이면 **Start Sensor Reading** 버튼이 활성화되고 **Stop Sensor Reading** 버튼이 비활성화된다. 그렇지 않으면 이들 버튼의 `Enabled` 속성이 바뀐다.

- **IsStartSensorReadingButtonEnabled**: 이 속성은 **Start Sensor Reading** 버튼의 `Enabled` 속성을 제어한다.
- **IsStopSensorReadingButtonEnabled**: 이 속성은 **Stop Sensor Reading** 버튼의 `Enabled` 속성을 제어한다.

앱을 실행하면 **Connect and Initialize** 버튼을 클릭할 수 있다. Sense HAT 애드온 보드가 있고 모든 부분이 올바르게 준비되었다면 **Start Sensor Reading** 버튼이 활성화된다. 이 버튼을 클릭하면 백그라운드 작업이 시작되고, 실제 센서 판독값은 **Stop Sensor Reading** 버튼을 클릭할 때까지 표시된다. 그림 D-7에서 관성 센서의 샘플 판독값을 나타냈다.

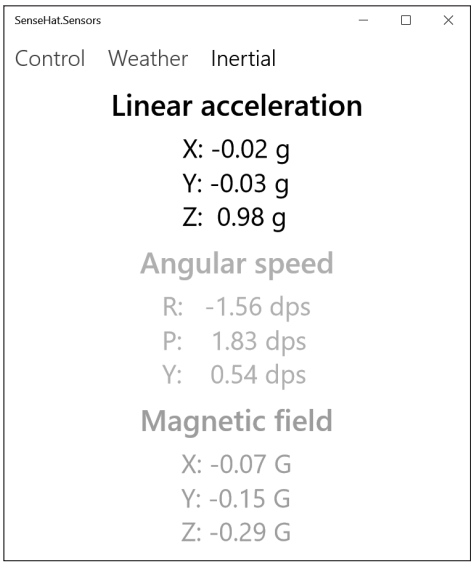

그림 D-7 SenseHat.Sensors 앱의 Inertial 탭

APPENDIX E

비주얼 C++ 구성 요소 확장

비주얼 C++ 구성 요소 확장(C++/CX)은 Windows 런타임과 네이티브 C/C++ 코드를 직접 인터페이스할 수 있는 언어 확장 집합을 구성한다. 8장, '이미지 처리'에서 설명한 이미지 처리 작업처럼 복잡한 작업과 기존 코드 및 라이브러리를 재사용할 수 있도록 C++/CX 구성 요소를 구축할 수 있다. 일반적으로 C++/CX 구성 요소를 Windows 런타임 구성 요소 또는 DLL로 생성한 다음 UWP 프로젝트에서 참조한다. 해당 프로젝트에서 다른 UWP 어셈블리와 동일한 방식으로 C++/CX 구성 요소에서 메서드를 호출한다. 이러한 방식으로 C++/CX는 네이티브 프로그래밍에 대한 액세스를 제공하는 동시에 C++ 프로그래밍의 최신 요소를 사용하고 UWP 솔루션의 상위 계층과 간단하게 통합할 기회를 제공한다.

전체 UWP 앱을 C++/CX로 작성할 수도 있다. 이는 C#보다 C++를 선호하거나 C++/CX 앱이 빌드 구성과 관계없이 네이티브 코드로 컴파일되기 때문에 향상된 속도 성능이 필요한 경우 특히 유용할 수 있다. 그러나 대부분의 경우 레거시legacy 코드를 인터페이스하고자 C++/CX 래퍼를 준비한 뒤 C#을 사용해 UI 계층과 함께 로직을 작성한다.

부록 E에서는 C++/CX로 UWP 앱 구축을 시작하는 방법을 보여 준다. 특히 백그라운드에서 주기적인 센서 판독 값을 에뮬레이션하는 앱을 구현하는 방법을 알려 준다. 이러한 백그라운드 작업은 이 책에서 다룬 가장 일반적인 기능 중 하나였다. 이 앱은 IoT 관점에서 C++/CX의 가장 중요한 요소를 설명한다. 자세한 내용과 전체 C++/CX 언어 참조는 https://bit.ly/cpp_cx에서 확인할 수 있다.

사용자 인터페이스 및 이벤트 처리

부록 E에서 다루는 소스 코드는 함께 제공되는 코드의 [Appendix E] 폴더 아래의 첨부했다. 비어 있는 앱(유니버설 Windows) 비주얼 C++ 프로젝트 템플릿을 사용해 SenseHat.Sensors.CppCx라는 하나의 프로젝트를 생성했다. 그런 다음 2개의 버튼과 2개의 레이블로 구성된 UI를 생성했다(그림 E-1 참고). 해당 버튼은 백그라운드 작업을 시작하고 중지하는 데 사용된다. 한 레이블은 상수 문자열 `Temperature`를 표시하고 다른 레이블은 에뮬레이트 된 센서 판독 값을 표시한다.

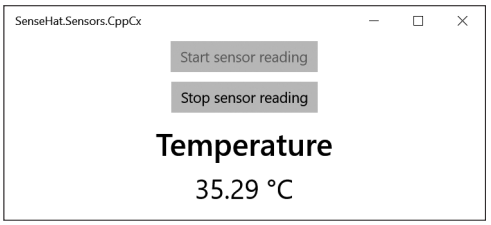

그림 E-1 SenseHat.Sensors.CppCx의 UI

UI 선언은 C# 앱과 동일한 XAML 구문을 사용한다(함께 제공되는 코드 [Appendix E/SenseHat.Sensors.CppCx/MainPage.xaml] 참고). 그러나 연결된 코드 숨김은 MainPage.xaml.h 및 MainPage.xaml.cpp의 두 파일로 구성된다. 첫 번째 파일은 `MainPage` 클래스 선언을 포함하고, 다른 파일은 해당 정의를 구현한다.

예제 E-1에서 볼 수 있듯이 1개의 public 읽기 전용 속성(`SensorsViewModel^` 유형의 `ViewModel`), 2개의 private 필드(`sensorsViewModel` 및 `telemetry`), 3개의 메서드(`OnDataReady`, `ButtonStartSensorReading_Click`, `ButtonStopSensorReading_Click`)로 `MainPage` 선언을 확장한다. 해당 선언을 분석할 때 가장 눈에 띄는 구문은 ^(삿갓표)다. 이 선언자는 스마트 포인터 역할을 한다. 런타임에 객체의 수명을 자동으로 관리하도록 지시하므로 참조 카운터가 0이 되면 해당 메모리가 해제된다. 또한 ^는 자동 유형 변환을 제공한다. 따라서 ^가 있는 유형을 일반적으로 관리 유형managed type이라고 한다.

예제 E-1 MainPage 클래스 선언

```
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);
};
```

MainPage에서 대부분의 형식은 UWP API와 관련돼 있으므로 ^로 선언된다. 그러나 C++/CX에서는 *로 선언된 표준 포인터를 사용할 수도 있다. 관리 유형은 ref new 키워드로 생성하는 반면 관리되지 않는 유형의 경우 일반적으로 new 키워드를 사용한다. 관리 유형의 멤버에 액세스하려면 멤버 접근 연산자(->)를 사용한다.

버튼 이벤트 핸들러의 정의는 예제 E-2에 나와 있다. 원격 분석을 시작 및 중지하고 ViewModel의 IsTelemetryActive 속성을 업데이트한다. 후자는 버튼 상태를 제어하고 에뮬레이트된 온도를 표시하고자 UI에 바인딩된다.

예제 E-3에서 보인 것처럼 온도 값은 Telemetry 클래스의 DataReady 이벤트를 사용해 전달된다. 예제 E-3의 코드 블록은 이벤트 핸들러를 이벤트와 연결하는 방법을 보여 준다. 이는 MainPage 생성자 내에서 수행되며, 여기서 += 연산자를 사용해 DataReadyEventHandler 대리자delegate 형식의 이벤트 처리기를 생성한다. 이 구문은 C#과 매우 비슷하다. 그러나 이벤트를 처리하는 실제 메서드를 나타내려면 &연산자를 사용한다.

예제 E-2 원격 측정 시작 및 중지

```
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;
}
```

예제 E-3 이벤트 처리

```
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);
        }));
    }
}
```

이 예제에서 이벤트는 OnDataReady 메서드에서 사용된다. 예제 E-3에서 보인 것처럼 이 메서드는 ViewModel의 해당 속성에 온도 값을 다시 작성한다. ViewModel은 UI에 바인딩돼 있기 때문에 자동으로 UI 업데이트를 트리거하므로 이 작업이 스레드로부터 안전한지 확인해야 한다. 이를 위해 C# 프로젝트와 비슷하게 사용되는 Dispatcher 객체를 사용한다. 여기서 한 가지 부분만 추가 설명이 필요하다. 이는 DispatchedHandler를 생성하는 부분이다. 여기서 캡처 조항capture clause [=]과 함께 람다 표현식(또는 간단히 람다, https://bit.ly/cpp_lambda)을 사용한다. 이 조항은 람다 표현식을 둘러싼 범위에서 람다 본문까지의 변수를 캡처하는 데 사용된다. 이 경우 모든 변수는 값으로 캡처된다. 참조로 변수를 캡처해야 하는 경우 [&] 조항을 사용한다. 특정 변수에 특정 캡처 유형을 사용할 수도 있다. 캡처 조항 내에 변수 이름을 작성하고, 참조 캡처를 위해 앞에 &를 붙인다. 빈 조항 []은 어떠한 변수도 캡처하지 않는다.

이벤트 선언 및 이벤트 인수

이전에 사용한 DataReadyEventHandler의 선언은 Telemetry 헤더 파일에 저장된다(함께 제공되는 코드 [Appendix E/SenseHat.Sensors.CppCx/Telemetry.h] 참고). 예제 E-4에서 보인 것처럼 DataReadyEventHandler에는 접근 한정자, delegate 키워드, 반환 유형, 형식 인수 목록이 포함돼어 있다. 이 선언문은 ^을 제외하고 C#과 동일하다.

예제 E-4 대리자 선언

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

DataReadyEventHandler에는 sender와 e의 두 가지 인수가 있다. 첫 번째 인자는 참조를 객체에 전달해 이벤트를 발생시키는 데 사용되며, 후자는 원격 측정 데이터를 저장한다. 이 경우 센서 판독 값은 TelemetryEventArgs 클래스에 래핑되며, 해당 선언은 예제 E-5에 표시된다(함께 제공되는 코드 [Appendix E/SenseHat.Sensors.CppCx] 참고).

예제 E-5 TelemetryEventArgs 클래스 선언

```
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;
};
```

TelemetryEventArgs는 1개의 private 필드인 temperature와 1개의 public 속성인 Temperature를 선언한다. 속성 선언에 따르면 Temperature 멤버는 공개적으로 접근할 수 있지만, 비공개로 수정할 수 있다. 이 경우 C# 구문은 훨씬 더 간단하며, 추가 private 필드를 선언할 필요가 없다.

예제 E-6에 표시된 TelemetryEventArgs의 생성자는 Temperature 속성을 설정하는 데 사용된다. Telemetry 클래스에서 이 생성자를 사용해 에뮬레이트된 온도를 TelemetryEventArgs의 인스턴스로 쉽게 래핑한다.

예제 E-6 TelemetryEventArgs 생성자

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

병렬 처리

백그라운드 작업을 구현하고자 Telemetry 클래스를 생성한다. 예제 E-7에서 보인 것처럼 이 클래스는 DataReady 이벤트와 이벤트를 발생시키는 OnDataReady 인라인 함수를 선언한다. C++/CX에서는 사용자 정의 이벤트 발생 로직을 작성하지 않고는 클래스 정의에서 이벤트를 직접 발생시킬 수 없기 때문에 이 추가 함수가 필요하다(https://bit.ly/cpp_cx_events 참고).

예제 E-7 Telemetry 클래스 선언

```
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();
};
```

백그라운드 작업이 진행 중인지 아닌지를 나타내는 플래그인 IsActive 속성을 선언한다. 또한 Start 및 Stop이라는 두 가지 public 메서드가 있다. 이 메서드는 msDelayTime 상수로 지정된 지연 시간에 센서 판독 값을 주기적으로 에뮬레이트하는 데 사용된다.

더 나아가 Telemetry 클래스는 telemetryTask 및 telemetryCancellationTokenSource의 두 private 필드를 선언한다. 각각 task<T> 및 cancel_token_source 유형이다. 두 가지 유형은 모두 병렬 패턴 라이브러리(PPL, Parallel Patterns Library; https://bit.ly/pp_lib)의 일부다. PPL은 Windows ThreadPool 내에서 병렬 작업을 실행하는 데 사용할 수 있는 API를 제공한다. 또한 PPL은 일반 병렬 알고리즘(예: 병렬 루프) 및 병렬 컨테이너를 구현한다.

예제 E-8에서 보인 것처럼 InitializeTelemetryTask 내에서 telemetryTask 및 telemetryCancellationTokenSource를 인스턴스화한다. 먼저 작업을 취소하는 데 사용되는 cancel_token_source를 생성한다. 참고로 cancel_token_source는 관리 유형이 아니므로 포인터 유형으로 선언되고, new 키워드로 생성된다. 일반 인수로 void를 사용해 작업 클래스를 인스턴스화한다. 여기에 사용된 람다는 캡처 조항([this])에서 현재 클래스 인스턴스를 검색한다. 작업 메서드 자체는 C# 예제와 비슷하다. 즉 센서 판독 값을 생성한 뒤 이벤트를 사용해 리스너에게 알린다. 이러한 작업은 Sleep 함수를 사용해 시간이 지연된다.

예제 E-8 원격 측정 작업 초기화

```
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());
}
```

센서 판독 값을 에뮬레이트하고자 예제 E-9에 나오는 GetSensorReading 메서드를 작성한다. 온도 변화를 에뮬레이트하고자 GetSensorReading은 섭씨 35도의 기본 온도에 추가되는 임의로 생성된 값을 사용한다.

예제 E-9 온도 판독 에뮬레이션

```
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);
}
```

원격 측정은 예제 E-10에 표시된 Start 메서드를 호출해 시작된다. 참고로, 작업을 초기화한 후에는 병렬 작업을 명시적으로 실행하는 추가 메서드를 호출할 필요가 없다. 예제 E-8의 작업 메서드는 자동으로 ThreadPool에 대기하고 시작된다. 따라서 InitializeTelemetryTask를 호출하기 전에 IsActive 플래그를 true로 설정한다.

예제 E-10 백그라운드 작업으로 원격 분석 시작

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

        InitializeTelemetryTask();
```

```
    }
}
```

백그라운드 작업은 예제 E-11의 Stop 메서드를 사용하면 중지된다. 이 메서드는 cancellation_token_source를 사용해 작업에 신호를 보내고, IsActive 플래그를 false로 설정한다. 또한 포인터 변수이기 때문에 cancellation_token_source를 사용해 리소스를 수동으로 해제해야 한다.

예제 E-11 작업에 취소 신호를 보내 원격 측정 중지하기

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

        IsActive = false;

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

데이터 바인딩

추가 논의가 필요한 SenseHat.Sensors.CppCx 앱의 마지막 요소는 뷰 모델의 속성을 UI에 바인딩하는 방법이다. XAML 선언은 변경되지 않은 상태로 유지된다. 그러나 로직에는 약간의 차이가 있다. 뷰 모델을 구현하는 클래스는 Windows::UI::Xaml::Data::BindableAttribute와 연결돼야 한다.

여기에서 뷰 모델은 SensorsViewModel 클래스 내에서 구현된다(함께 제공되는 코드 [Appendix E/SenseHat.Sensors.CppCx] 참고). 이 선언은 예제 E-12에 나타난다. C#에서와

마찬가지로 SensorsViewModel 클래스는 INotifyPropertyChanged 인터페이스를 구현한다. 따라서 SensorsViewModel은 PropertyChangedEventHandler 유형의 이벤트 PropertyChanged를 선언한다. 이 이벤트는 OnPropertyChanged 메서드 내에서 발생한다. 기본적으로 C#과 같아 보이지만 C++/CX에서는 CallerMemberNameAttribute를 사용할 수 없으므로 속성 이름을 OnPropertyChanged 메서드에 명시적으로 전달해야 한다. 특히 OnPropertyChanged는 Temperature 및 IsTelemetryActive 속성 설정에서 호출된다. 그럴 때마다 데이터 바인딩 메커니즘은 PropertyChanged 이벤트를 처리해 UI의 해당 요소를 업데이트한다. 온도 값은 Temperature 속성을 통해 UI에 표시되고, IsTelemetryActive는 센서 판독 시작Start Sensor Readings 및 중지Stop Sensor Readings 버튼을 활성화 또는 비활성화한다.

예제 E-12 SensorsViewModel 클래스 선언

```
[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));
        }
};
```

SensorsViewModel의 정의는 예제 E-13에 나와 있으며, 생성자와 ToggleStartStopButtons 메서드를 포함한다. 후자는 IsStartSensorReadingButtonEnabled 및 IsStopSensorReadingButtonEnabled를 설정해 UI의 현재 원격 분석 상태를 반영한다. 따라서 시작 시 Start Sensor Readings 버튼만 활성화된다(예제 E-13의 SensorsViewModel 생성자 참고).

예제 E-13 SensorsViewModel 클래스의 정의

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

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

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

값 변환기

C#에서와 같이 데이터 바인딩을 통해 전송된 데이터를 변환하는 값 변환기value converter를 사용할 수도 있다. 여기서는 이러한 변환기를 구현해 UI에 표시되는 온도를 형식화한다. 예제 E-14에 나오는 TemperatureToStringConverter의 선언은 값 변환기 클래스가 IValueConverter 인터페이스를 구현해야 함을 보여 준다. 따라서 TemperatureToStringConverter에는 Convert와 ConvertBack이라는 두 가지 메서드가 있다. 해당 부분의 의미는 이전 C# 예제와 동일하다.

예제 E-14 변환기 선언

```
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);
};
```

실제 변환을 수행하고 온도 문자열을 준비하고자 표준 C/C++ 기법을 사용한다(예제 E-15 참고). 먼저 static_cast 템플릿을 사용해 Object^를 double로 변환한다. 그런 다음 sprintf_s 함수를 사용해 온도를 형식화하고 °C를 추가한다. 이 작업의 결과는 wchar_t(와이드 문자 유형)의 배열에 저장된다. 마지막으로 해당 클래스의 전용 생성자를 사용해 이 배열을 String^으로 변환한다. 이는 관리 코드를 네이티브 코드와 얼마나 쉽게 혼용할 수

있는지 보여 주는 또 다른 예다. 그러나 C#과 비교해서 앱 로직을 작성하는 데 시간이 소요된다.

예제 E-15 변환기 정의

```
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();
}
```

변환기를 사용하려면 컨트롤, 페이지 또는 앱 리소스에서 선언해야 한다. 이 예제에서는 앱 리소스에 `TemperatureToStringConverter`를 선언하므로 예제 E-16과 같이 App.xaml 파일을 수정한다. 앱 리소스에서 `TemperatureToStringConverter`를 표시하려면 App.xaml.h에 적절한 헤더 파일을 포함해야 한다.

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

예제 E-16 애플리케이션 리소스의 변환기 선언

```
<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>
```

요약

부록 E에서는 C++/CX로 UWP IoT 앱을 구현하는 데 유용한 C++/CX의 기능을 설명했다. 여기서 개발된 기능은 센서의 데이터를 주기적으로 읽는 데 사용된다. 이 예제에서는 에뮬레이션된 판독 값을 사용했지만, 해당 값을 Sense HAT 애드온 보드에서 얻은 실제 데이터로 쉽게 대체할 수 있다.

APPENDIX F

IoT 개발을 위한 비주얼 스튜디오 2019 설정

이 책을 번역하는 동안 비주얼 스튜디오 2019가 출시됐다. 이 책에서 원래 다뤘던 비주얼 스튜디오 2015와는 다른 인스톨러를 제공하기 때문에 비주얼 스튜디오 2019를 설치하는 방법과 이 책 전체에서 소개하는 샘플 앱을 구현하는 데 사용하는 방법을 설명한다. 덧붙여 IoT 디바이스에 C# 앱을 설정하고 배포하는 방법과 비주얼 스튜디오 2019를 사용해 이식 가능한 클래스 라이브러리(PCL)를 구성하는 방법을 설명한다.

설치

그림 F-1에서 비주얼 스튜디오 2019 인스톨러를 나타냈다.

그림 F-1 비주얼 스튜디오 2019 인스톨러

이 책에서 IoT 개발을 위해 **유니버설 Windows 플랫폼 개발** 도구(그림 F-2 참고)와 **C++(v142) 유니버설 Windows 플랫폼 도구**가 필요하다.

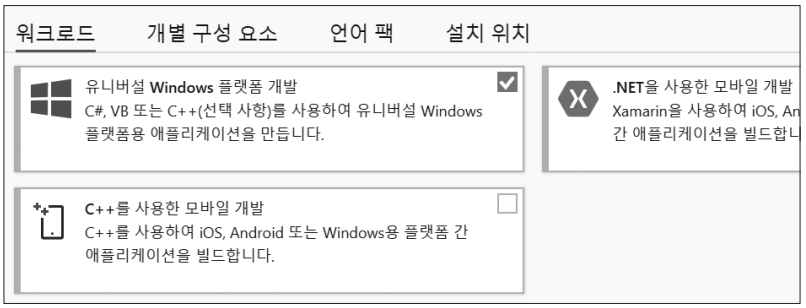

그림 F-2 유니버설 Windows 플랫폼 개발 워크로드

그림 F-3 C++ UWP 지원 패키지 설치

추가로 UWP용 IoT 확장을 지원하고자 Windows 10 SDK(10.0.17134.0)를 선택해 설치한다.

비주얼 스튜디오 2019용 Windows IoT 코어 프로젝트 템플릿을 설치하고자 **확장 > 확장 관리** 메뉴를 열고 **온라인** 탭을 클릭한 다음 검색 상자에 **IoT**를 입력한다. 잠시 뒤에 **Windows IoT Core Project Templates for VS 2017+** 옵션이 표시된다(그림 F-4 참고).

그림 F-4 비주얼 스튜디오 2019를 지원하는 윈도우 IoT 코어 프로젝트 템플릿

비주얼 C# 프로젝트 템플릿

모든 도구가 준비되면 비주얼 C# 프로젝트 템플릿을 사용해 UWP 앱을 작성한 다음 IoT 디바이스에 배포할 수 있다.

프로젝트 만들기

비주얼 스튜디오 2015에서 **새 프로젝트** 대화 상자(**파일** 메뉴를 열고 **새 프로젝트** 선택)를 사용했던 방식과 비슷하게 프로젝트를 만든다. 템플릿 검색 상자에서 **비어 있는 앱**을 입력하고 C#용 **비어 있는 앱(유니버설 Windows)**를 선택하고 프로젝트와 솔루션 이름을 설정한다(그림 F-5 참고).

노트 | 비주얼 스튜디오 2015와 달리 [새 프로젝트 만들기] 대화 상자는 [추가 도구 및 기능 설치] 하이퍼링크를 포함한다. 이 링크를 사용해 [새 프로젝트 만들기] 대화 상자에 없는 기능과 템플릿을 추가할 수 있는 인스톨러 창을 열 수 있다.

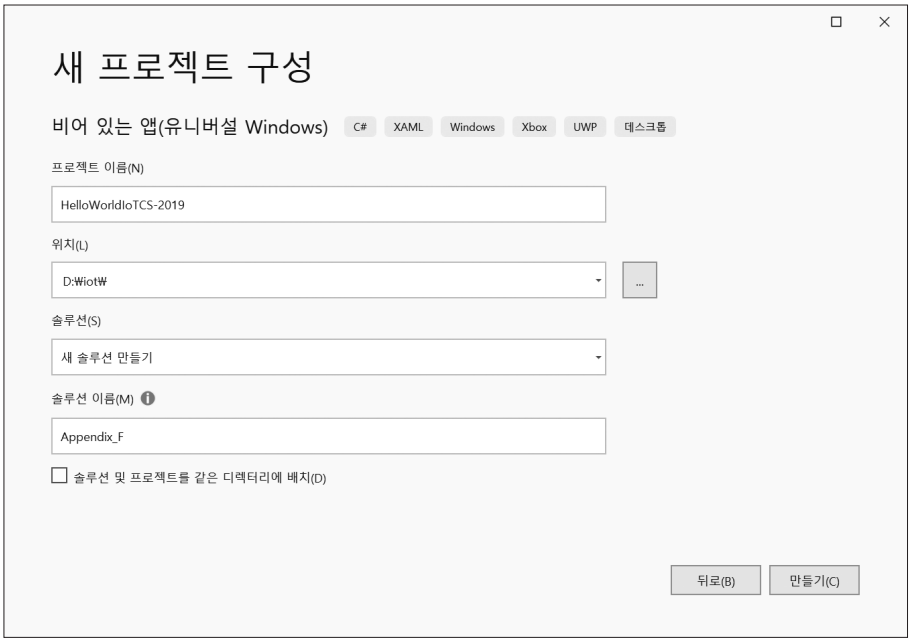

그림 F-5 비주얼 스튜디오 2019의 [새 프로젝트 구성] 대화 상자

만들기 버튼을 클릭한다. 대상 및 최소 플랫폼 버전을 선택할 기회가 주어진다. 그림 F-6에서 보인 것처럼 이 대화 상자는 비주얼 스튜디오 2015 업데이트 3과 정확히 동일하다. 대상 버전을 **Windows 10, version 1903(10.0, 빌드 18362)**로 하고, 최소 버전을 **Windows 10, version 1809(10.0, 빌드 17134)**로 설정한다.

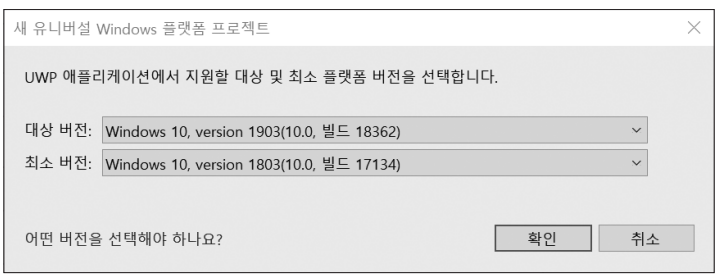

그림 F-6 대상 및 최소 플랫폼 버전 선택

IoT 확장

UWP의 IoT 전용 API를 액세스하려면 **Windows IoT Extensions for the UWP**를 참조해야 한다. 이 과정은 비주얼 스튜디오 2015와 비슷하다. **솔루션 탐색기**에서 **참조** 노드를 오른쪽 클릭한 다음 [참조 추가]를 선택한다. 다음으로 **Universal Windows ➤ 확장** 탭을 클릭하고 **Windows IoT Extensions for the UWP(10.0.17134.0)**을 선택한다(그림 F-7 참고).

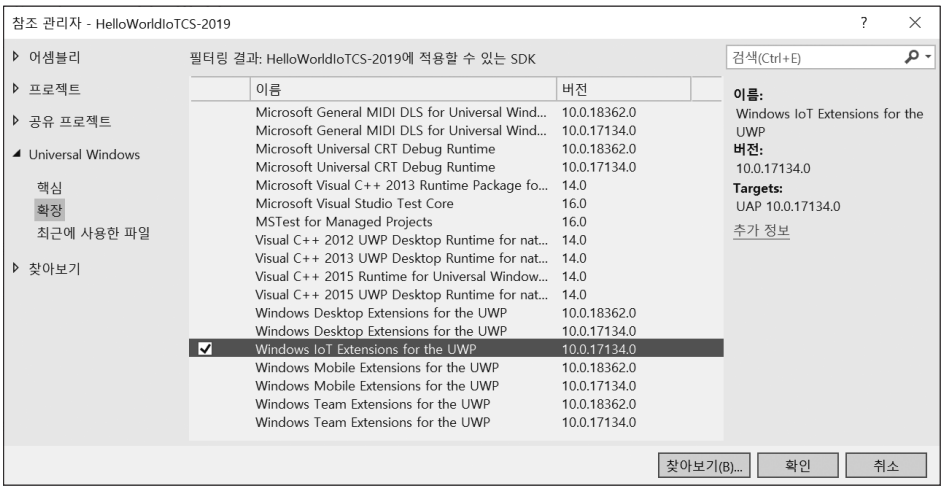

그림 F-7 비주얼 스튜디오 2019의 [참조 관리자]

구현

먼저 `LedBlinking` 클래스를 만들어 HelloWorldIoTCS-2019 앱을 구현한다(Appendix F/HelloWorldIoTCS-2017/GpioControl/LedBlinking.cs에서 함께 제공하는 코드 참고). 이 클래스는 지정한 시간 간격으로 선택된 GPIO 핀을 반전시키는 백그라운드 작업을 시작하고 중지하는 데 사용된다. 이 기능을 사용해 RPi2의 보드에 포함된 녹색 LED를 제어한다. RPi3 이상에서 외부 LED를 제어하려면 적절한 GPIO 핀 번호를 사용해야 한다.

`LedBlinking`은 내부적으로 2장, '디바이스용 유니버설 Windows 플랫폼'과 3장, '윈도우 IoT 프로그래밍 에센셜'에서 기술한 메서드들을 사용한다. `LedBlinking` 클래스의 주 요소는 예제 F-1의 `InitializeBlinkingTask` 메서드다. 이 메서드는 주기적으로 GPIO 핀 상

태를 반전시켜 LED를 제어한다. 깜박임을 시작하고 중지하고자 LedBlinking 클래스의 Start와 Stop 메서드를 구현한다.

예제 F-1 깜박임을 수행하는 백그라운드 작업 초기화

```
public int MsShineDuration { get; private set; }

private Task blinkingTask;
private CancellationTokenSource blinkingCancellationTokenSource;

private GpioPin gpioPin;

private void InitializeBlinkingTask()
{
    blinkingCancellationTokenSource = new CancellationTokenSource();

    blinkingTask = new Task(() =>
    {
        while (!blinkingCancellationTokenSource.IsCancellationRequested)
        {
            if (IsActive)
            {
                SwitchGpioPin(gpioPin);

                Task.Delay(MsShineDuration).Wait();
            }
        }
    }, blinkingCancellationTokenSource.Token);
}
```

예제 F-2 깜박임을 수행하는 백그라운드 작업 시작 및 중지

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

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

        blinkingTask.Start();
```

```
        IsActive = true;
    }
}

public void Stop()
{
    if (IsActive)
    {
        blinkingCancellationTokenSource.Cancel();

        IsActive = false;
    }
}
```

LedBlinking 클래스가 준비됐으므로 UI 작업을 시작한다. 여기서는 Initialize, Start Blinking, Stop Blinking이라는 세 가지 버튼으로 구성된 간단한 UI를 정의한다(그림 F-8 참고).

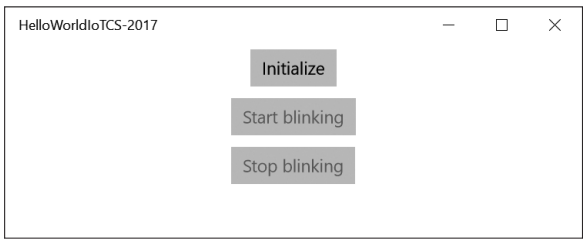

그림 F-8 HelloWorldIoTCS-2019 앱의 사용자 인터페이스

첫 번째 버튼인 Initialize는 LedBlinking 클래스 인스턴스를 생성하는 데 사용된다(예제 F-3 참조). 이 이벤트 핸들러는 BlinkingViewModel의 GpioPinNumber와 MsShineDuration 속성을 사용한다. 이들 속성은 GPIO 핀 번호와 LED 깜박임 주파수(GPIO 핀 스위칭 시간 간격)를 변경하는 데 사용된다. 이들 속성의 기본값은 예제 F-4에서 나타냈다.

예제 F-3 LedBlinking 클래스의 인스턴스 생성

```
private BlinkingViewModel blinkingViewModel = new BlinkingViewModel();
private LedBlinking ledBlinking;
```

```
private void ButtonInitializeGpio_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ledBlinking = new LedBlinking(blinkingViewModel.GpioPinNumber,
            blinkingViewModel.MsShineDuration);
        blinkingViewModel.IsGpioPinAvailable = true;
    }
    catch (Exception ex)
    {
        Debug.WriteLine(ex.Message);
    }
}
```

예제 F-4 GPIO 핀 번호와 LED 깜박임 주파수는 BlinkingViewModel의 적절한 속성을 통해 구성된다.

```
public int GpioPinNumber { get; set; } = 47;
public int MsShineDuration { get; set; } = 100;
```

초기화 성공 후 `BlinkingViewModel` 클래스(Appendix F/HelloWorldIoTCS-2017/ViewModels/BlinkingViewModel.cs에서 함께 제공하는 코드 참고)의 `IsGpioPinAvailable` 속성과 `LogicalNegationConverter`(Appendix F/HelloWorldIoTCS-2019/Converters/LogicalNegationConverter.cs에서 함께 제공하는 코드 참고)를 통해 해당 버튼을 비활성화한다. Initialize 버튼은 `IsGpioPinAvailable`이 `false`일 때만 활성화돼야 하기 때문에 이 변환기를 사용한다. 따라서 `LogicalNegationConverter`는 부울 값을 부정한다. 전체 앱에서 이 변환기를 사용하고자 예제 F-5에서 나타낸 것처럼 App.xaml 파일을 수정한다.

예제 F-5 애플리케이션 범위 리소스에서 LogicalNegationConverter 포함

```
<Application
    x:Class="HelloWorldIoTCS_2017.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:converters="using:HelloWorldIoTCS_2017.Converters"
    RequestedTheme="Light">

    <Application.Resources>
```

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

BlinkingViewModel의 IsGpioPinAvailable 속성은 LED를 제어하는 데 사용되는 GPIO 핀이 사용 가능한지의 여부를 가리킨다. 사용 가능하다면 LED 깜박임의 백그라운드 작업이 시작될 수 있고, 그다음 Start Blinking과 Stop Blinking 버튼을 각각 클릭해 시작하고 중지할 수 있다. 이들 버튼이 클릭될 때마다 이벤트 핸들러는 LedBlinking 클래스의 해당 메서드를 호출한다(예제 F-6 참고).

예제 F-6 LED 깜박임 시작 및 중지

```
private void ButtonStartBlinking_Click(object sender, RoutedEventArgs e)
{
    ledBlinking.Start();
    blinkingViewModel.IsBlinkingActive = ledBlinking.IsActive;
}

private void ButtonStopBlinking_Click(object sender, RoutedEventArgs e)
{
    ledBlinking.Stop();
    blinkingViewModel.IsBlinkingActive = ledBlinking.IsActive;
}
```

LED 깜박임의 상태(활성 또는 비활성)는 BlinkingViewModel의 IsBlinkingActive 속성으로 가리킨다. 따라서 IsBlinkingActive는 LedBlinking.Start 호출 후 true로 바뀌고, LedBlinking.Stop 호출 후 false로 바뀐다. BlinkingViewModel의 클래스 인스턴스의 IsBlinkingActive 속성을 설정하고자 LedBlinking.IsActive 플래그를 사용한다.

본질적으로 데이터 바인딩을 위해 LedBlinking.IsActive 속성을 사용할 수 있다. 하지만 BlinkingViewModel.IsBlinkingActive의 설정 내에서 추가 로직을 수행한다. 즉 예제 F-7에서 보인 것처럼 PropertyChanged 이벤트를 일으켜 속성 변경에 관해 UI에 통지한 다음 BlinkingViewModel 속성(IsStartBlinkingButtonEnabled와 IsStopBlinkingButtonEnabled)을 토글해 **Start Blinking**과 **Stop Blinking** 버튼의 상태를 제어한다(예제 F-8 참고).

예제 F-7 BlinkingViewModel의 IsBlinkingActive 속성 정의

```
private bool isBlinkingActive;

public bool IsBlinkingActive
{
    get { return isBlinkingActive; }
    set
    {
        isBlinkingActive = value;
        OnPropertyChanged();

        ToggleStartStopButtons(!value);
    }
}
```

예제 F-8 [Start Blinking]과 [Stop Blinking] 버튼의 Enabled 속성은 데이터 바인딩을 통해 제어된다.

```
public bool IsStartBlinkingButtonEnabled { get; private set; }
public bool IsStopBlinkingButtonEnabled { get; private set; }

private void ToggleStartStopButtons(bool isStartEnabled)
{
    IsStartBlinkingButtonEnabled = isStartEnabled;
    OnPropertyChanged("IsStartBlinkingButtonEnabled");

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

솔루션 구성 및 배포

라즈베리 파이 2와 파이 3에 앱을 배포하고자 2장에서 설명한 것처럼 진행한다. 즉 솔루션 플랫폼을 ARM으로 설정하고 대상을 **원격 컴퓨터**로 변경한다(그림 F-9 참고). 이 작업을 처음 수행한다면 그림 F-10의 **원격 연결** 대화 상자가 나타난다. 이 대화 상자는 나중에 프로젝트 속성(**디버그** 탭을 찾은 다음 **시작 옵션** 그룹에서 **찾기** 버튼 클릭)을 사용해 열 수 있다.

그림 F-9 컴파일, 솔루션 플랫폼, 대상 구성

원격 디바이스를 선택한 후 **디버그** 메뉴를 열고 **디버깅 시작** 또는 **디버그하지 않고 시작**을 선택하거나 **원격 컴퓨터** 버튼(그림 F-9 참고)을 클릭해 앱을 실행한다. 디버깅 없이 앱을 실행하면 디바이스 포털의 **Processes** 탭에서 앱을 중지할 수 있다.

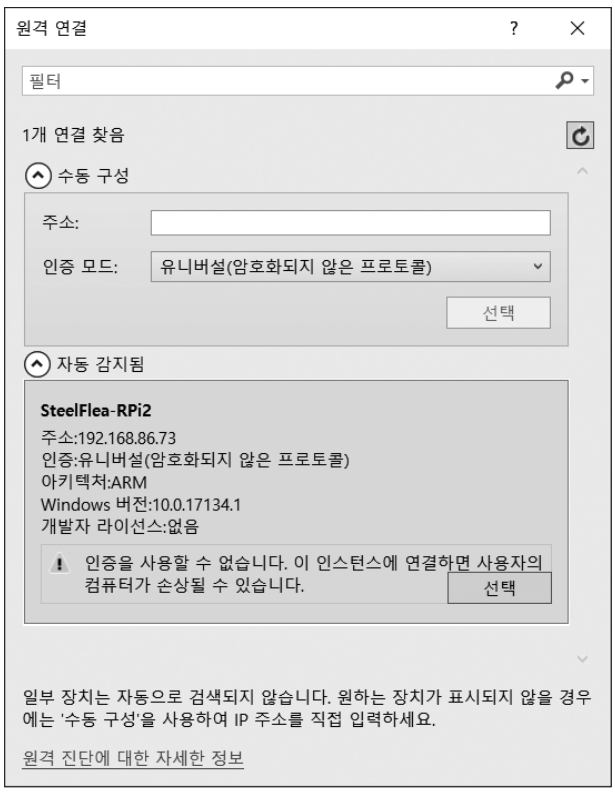

그림 F-10 로컬 네트워크에서 윈도우 원격 컴퓨터를 찾는 데 사용하는 원격 연결 대화 상자

UI 없는 프로젝트 템플릿은 비주얼 스튜디오 2015 업데이트 2와 3에서 동일하기 때문에 UI 있는 앱을 작성하는 방법만 보였다.

이식 가능한 클래스 라이브러리

이식 가능한 클래스 라이브러리PCL, Portable Class Library 프로젝트를 만들고자 C#용 **클래스 라이브러리(.NET Standard)** 프로젝트 템플릿을 사용한다. 이 템플릿은 **새 프로젝트 만들기** 대화 상자의 템플릿 검색 상자에서 **클래스 라이브러리**를 입력해 찾을 수 있다. 이식 가능한 클래스 라이브러리를 만드는 방법의 자세한 내용은 부록 D를 참고하기 바란다.