Introducing Devices Profile for Web Services

Devices Profile for Web Services (DPWS) is a Web Services profile that enables plug-and-play for networked devices. A PC or other device can detect DPWS-enabled devices on a network, then discover and invoke the Web service functionality each device provides. Its purpose is similar to Universal Plug and Play (UPnP), although it employs a Web Services model.

DPWS, also called Web Services on Devices (WSD), is part of the Windows Rally technology set for network-connected devices.[1] A DPWS client library (known as WSDAPI) is a part of Windows Vista, and DPWS-enabled devices automatically appear in the Windows Vista Network Explorer. And with version 2.5, DPWS is also part of the .NET Micro Framework, so you can provide or consume DPWS functionality in small devices.

To help you get a head start on DPWS, we have prepared a discussion based on an early draft of the documentation that will be provided with version 2.5 of the .NET Micro Framework. To get the most from this article, you should already understand basic Web Services concepts. MSDN has a good introduction[2] and numerous other resources on the topic.

NOTE The DPWS implementation in .NET Micro Framework v2.5 should be considered a technology preview. The API will likely change significantly in a future release, which will also incorporate tools that will simplify writing services and make parsing and generating messages easier. You will find the current implementation useful if you have a pressing need for DPWS functionality or if you simply want to experiment with Web Services.

Web Services Profiles and DPWS

In Web Services terms, a *profile* is a set of guidelines for how to use Web Services technologies for a given purpose or application. Web Services standards allow implementers to choose from a variety of message representations, text encodings, transport protocols, and other options, some of which are not interoperable. By constraining these decisions, profiles ensure that conforming implementations will work well together.

DPWS is a profile developed by Microsoft and others for communication with and among networked devices and peripherals. As such, the DPWS library for the .NET Micro Framework is not a full Web Services implementation but a lightweight subset with only the functionality needed to support DPWS on a device. The full DPWS profile specification is available on the Web.[3]

[1] http://www.microsoft.com/whdc/rally/default.mspx
[2] http://msdn2.microsoft.com/en-us/webservices/aa740678.aspx
[3] http://schemas.xmlsoap.org/ws/2006/02/devprof/

Communicating Between Devices

DPWS provides the following functionality between conforming devices:

- Discovering DPWS-capable devices on the network and the services they offer
- Sending messages to DPWS-capable devices and receiving replies
- Describing a Web service by providing a WSDL file
- Interacting with a service using its description
- Subscribing to and receiving events from a Web service

Devices can be DPWS clients (invoking services on devices), servers (providing services), or both. DPWS for the .NET Micro Framework supports devices in either role or both simultaneously. These two sets of functionality are provided in separate DLLs and are not dependant on each other.

DPWS is based on existing Web Services standards, including XML, WSDL, XML Schema, SOAP, MTOM, and HTTP. DPWS for the .NET Micro Framework also incorporates this supporting functionality, although not all of it is exposed in the APIs. For example, there are classes for reading and writing XML documents so that applications and service implementations can parse Web Services messages and build responses to them.

Implementation Specifics

The DPWS specification defines the required behavior of a DPWS implementation, but leaves some decisions to the implementer. The following list describes the behavior of the .NET Micro Framework DPWS implementation with respect to the specification.

- URIs longer than `MAX_URI_SIZE` (2,048 octets) are not processed and a SOAP fault is generated. (Section 3.1, requirement R0025.)
- The `address` property of a device's Endpoint Reference must be a UUID. (Section 3.5, requirement R0004.)
- A device will not include any reference properties in its Endpoint Reference. (Section 3.5, requirement R0007.)
- A service will always use an HTTP transport address as the address property of its Endpoint Reference. (Section 3.5, requirement R0042.) The address is formed using the device's IP address and a unique UUID-based URN.
- The developer must manually increment the metadata version number if any of the device's `ThisDevice` or relationship metadata changes. (Section 5.1 requirement R2002, Section 5.2 requirement 2030).
- The implementation does not provide any WSDL-related functionality aside from message validation (Section 5.3, requirements R2023 and R2024), although validation uses the .NET service classes rather than WSDL. The device developer must provide a WSDL description of their service and make it publicly accessible via URL (for example, by placing it on a Web server).

- This implementation does not support WS-POLICY (Section 5.4), DateTime expiration types (Section 6.2), or security (Section 7), all optional features.

The DPWS Libraries

The .NET Micro Framework DPWS device stack libraries, `MFWsStack.dll` and `MFDpwsDevice.dll`, provide service hosting functionality for a device (the "device stack"). The DPWS client library, `MFDpwsClient.dll`, provides functionality for invoking Web services on a device. A separate library, `MFDpwsExtensions.dll`, contains the System.Ext.* namespaces, which are partial implementations of certain .NET System.* classes required by the DPWS device stack that are not included in the base .NET Micro Framework libraries.

Since not all applications need DPWS functionality, these libraries are not automatically added to .NET Micro Framework projects you create in Visual Studio. Instead, you should manually add references to the DLLs to your project. To do this, follow these steps:

1. Right-click the References folder in your Visual Studio project.
2. Choose **Add Reference** from the context menu.
3. In the Add Reference dialog, choose the .NET tab.
4. Select `MFWsStack.dll` in the list, then click **OK**.
5. Repeat steps 1 to 3, this time adding `MFDpwsExtensions.dll`.
6. If your project is a for a device hosting a service, repeat steps 1 to 3, this time adding `MFDpwsDevice.dll`.
7. If your project requires DPWS client functionality, repeat steps 1 to 3, this time adding `MFDpwsClient.dll`.

DPWS Namespaces

The DPWS device stack DLLs (`MFDpwsDevice.dll` and `MFWsStack.dll`) include the following namespaces.

`Dpws.Device`	Provides a class that represents the DPWS device stack, and classes that hold properties that are used by the DPWS stack
`Dpws.Device.Services`	Contains classes used to create and manage event sources, and classes for working with hosted service endpoints, operations and events.
`Ws.Services`	Provides classes for collections of service endpoints managed by a transport host, and the operations they expose.
`Ws.Services.Faults`	Contains classes used to build and raise fault response messages.
`Ws.Services.Mtom`	Contains classes used to process MTOM parts in SOAP messages.
`Ws.Services.Soap`	Provides a class and an enumerated type for parsing SOAP messages.
`Ws.Services.Transport`	Contains a class that provides basic network services.
`Ws.Services.Utilites`	Provides classes for displaying debug output, representing time durations, and validating URIs based on UUIDs.
`Ws.Services.WsaAddressing`	Contains classes used to process WS-Addressing header information.
`Ws.Services.Xml`	Contains classes used to process XML.

The DPWS client DLL (MFDpwsClient.dll) includes the following namespaces.

Dpws.Client	Provides classes for creating DPWS clients.
Dpws.Client.Discovery	Contains classes for working with client probe and resolve requests.
Dpws.Client.Eventing	Contains classes for subscribing to events.
Dpws.Client.Transport	Contains classes for sending and receiving HTTP requests.

Using the Device Class

The static class Dpws.Device.Device represents the device stack itself. This class provides infrastructure functionality for DPWS-enabled devices as well as acting as the host for services you write. Include a call to Dpws.Device.Device.Start in your application's Main method to start the DPWS device stack.

The Device class also serves to contain parameters used by the device stack, such as the device's endpoint address. The Device.ThisModel class contains information specific to the device model, such as the model name and model number, and the Device.ThisDevice class contains information specific to the device itself, such as the serial number. Your Main method should set these as appropriate, using code like the following.

```
// Set device information (in Main)
Device.EndpointAddress = "http://localhost:1234";
Device.ThisModel.Manufacturer = "Microsoft Corporation";
Device.ThisModel.ManufacturerUrl = "http://www.microsoft.com/";
Device.ThisModel.ModelName = "SampleService Test Device";
Device.ThisModel.ModelNumber = "12021345";
Device.ThisModel.ModelUrl = "http://www.microsoft.com/";
Device.ThisModel.PresentationUrl = "http://www.microsoft.com/";

Device.ThisDevice.FriendlyName = "SampleService Device";
Device.ThisDevice.FirmwareVersion = "alpha";
Device.ThisDevice.SerialNumber = "12345678";

// Add a service
Device.HostedServices.Add(typeof(TestService), typeof(ITestService), new
       Uri("http://localhost:1234/TestService/"));

// Start the device stack
Device.Start();
```

Defining Services and Operations

The Dpws.Device.Device.HostedServices collection includes all services hosted by the device. Each service is derived from the class Dpws.Device.Services.DpwsHostedService.

Call the Add method in your application's Main to instantiate your services and add them to the HostedServices collection, as shown here. Any number of hosted services can be added. Services should be added before the device stack is started.

```
// Add hosted service to the device (in Main)
SampleService sampleService = new SampleService();
Device.HostedServices.Add(sampleService);
```

Service classes must be derived from the `DpwsHostedService` base class, which is used to define the service's transport address, unique identifier, namespace, and type. A service provides one or more *operations,* which are listed in the service class's `ServiceOperations` collection. Operations are methods of the service class, not classes of their own. As shown in the following code, the service's constructor should add the operations to the service's `ServiceOperations` collection in much the same way that services are instantiated and added to the `HostedServices` collection. The `SampleService` class also needs methods for the two defined operations, `OneWay` and `TwoWayRequest`.

```
class SampleService : DpwsHostedService
{
        public SampleService()
        {
              // Add ServiceNamespace. Set ServiceID and ServiceTypeName
              ServiceNamespace = new WsXmlNamespace("smpl",
                      "http://schemas.example.org/SampleService");
             ServiceID = "urn:uuid:3cb0d1ba-cc3a-46ce-b416-212ac2419b90";
             ServiceTypeName = "SampleService";

             // Add additional namespaces if needed, for example:
             // Namespaces.Add("someprefix", "http://some/Namespace");

             // Add service operations
             ServiceOperations.Add(new WsdHostedServiceOperation("smpl",
             "http://schemas.example.org/SampleService", "OneWay"));
                    ServiceOperations.Add(new WsdHostedServiceOperation("smpl",
                    "http://schemas.example.org/SampleService", "TwoWayRequest"));

             }

             // method for the first defined operation
             public byte[] OneWayRequest(WsWsaHeader header, WsXmlDocument envelope)
             {
                    ...
             }

             // method for the second defined operation
             public byte[] TwoWayRequest(WsWsaHeader header, WsXmlDocument envelope)
             {
                    ...
             }
       }
}
```

Operations receive two parameters: a `Ws.Services.WsaAddressing.WsWsaHeaderobject` containing validated header information, and a `Ws.Services.Xml.WsXmlDocument` object containing the entire SOAP request as an XML tree. The operation implementation must use methods of `WsXmlDocument` (for example, `SelectSingleNode`) to extract any values from the content of the message needed to perform the desired operation, and must return a byte array containing the SOAP response message for the request. The response is typically built using the original request header object, information provided by the method implementation, and a `System.Ext.XML.XMLWriter` object. (The `System.Ext.Xml` namespace is not a part of the base .NET Micro Framework, but is provided in `MFDpwsExtensions.dll`.)

The following code shows how parameters can be extracted from the SOAP message for an operation that adds two integers.

```
public byte[] TwoWayRequest(WsWsaHeader header, WsXmlDocument envelope)
{
	WsXmlNode tempNode;
	WsFault fault = new WsFault();

	if ((tempNode =
		envelope.SelectSingleNode("Body/TwoWayRequest/X", false)) == null)
			return fault.RaiseFault(header, WsExceptionFaultType.XmlException,
				"Body/TwoWay X value is missing.");

	int X = Convert.ToInt32(tempNode.Value);
	if ((tempNode =
		envelope.SelectSingleNode("Body/TwoWayRequest/Y", false)) == null)
			return fault.RaiseFault(header, WsExceptionFaultType.XmlException,
				"Body/TwoWay Y value is missing.");
	int Y = Convert.ToInt32(tempNode.Value);

	return TwoWayResponse(header, X+Y);
}
```

The response to a Web Services message is another message. It is often good practice to create a separate method that generates the response and call it from the method that handles the original request, particularly if either procedure is involved, or if similar responses are needed for more than one request. For example, the `TwoWayRequest` method (defined as part of the `SampleService` class partially shown above) might call a `TwoWayResponse` method to generate its response. Since the response method is called by the request handler and not by the DPWS stack, the response method is not limited to receiving a header and a SOAP request as parameters, but can receive any values needed to construct the response.

The following code shows an example of how the XML might be built in the `TwoWayResponse` method that is called from the example `TwoWayRequest` method.

```
public byte[] TwoWayResponse(WsWsaHeader header, int sum)
{
	MemoryStream soapStream = new MemoryStream();
	XmlWriter xmlWriter = XmlWriter.Create(soapStream);

	// Write processing instructions and root element
	xmlWriter.WriteProcessingInstruction("xml",
		"version='1.0' encoding='UTF-8'");
	xmlWriter.WriteStartElement("soap", "Envelope",
		"http://www.w3.org/2003/05/soap-envelope");

	// Write namespaces
	xmlWriter.WriteAttributeString("xmlns", "wsdp", null,
	Device.Namespaces.GetNamespace("wsdp"));
	xmlWriter.WriteAttributeString("xmlns", "wsd", null,
	Device.Namespaces.GetNamespace("wsd"));
	xmlWriter.WriteAttributeString("xmlns", "wsa", null,
	Device.Namespaces.GetNamespace("wsa"));
	xmlWriter.WriteAttributeString("xmlns", "sim", null,
	TypeNamespaces.GetNamespace("sim");


	// Write header
	xmlWriter.WriteStartElement("soap", "Header", null);
	xmlWriter.WriteStartElement("wsa", "To", null);
	xmlWriter.WriteString(header.From);
	xmlWriter.WriteEndElement();
```

```
        xmlWriter.WriteStartElement("wsa", "Action", null);
        xmlWriter.WriteString(
               "http://schemas.example.org/SimpleService/TwoWayResponse");
        xmlWriter.WriteEndElement();
        xmlWriter.WriteStartElement("wsa", "RelatesTo", null);
        xmlWriter.WriteString(header.MessageID);
        xmlWriter.WriteEndElement(); // End RelatesTo
        xmlWriter.WriteStartElement("wsa", "MessageID", null);
        xmlWriter.WriteString("urn:uuid:" + Device.GetUuid());
        xmlWriter.WriteEndElement(); // End MessageID
        xmlWriter.WriteEndElement(); // End Header

        // write body
        xmlWriter.WriteStartElement("soap", "Body", null);
        xmlWriter.WriteStartElement("sim", "TwoWayResponse", null);
        xmlWriter.WriteStartElement("sim", "Sum", null);
        xmlWriter.WriteString(sum.ToString());
        xmlWriter.WriteEndElement(); // End Sum
        xmlWriter.WriteEndElement(); // End TwoWayResponse
        xmlWriter.WriteEndElement(); // End Body

        xmlWriter.WriteEndElement();

        // Create return buffer and close writer
        xmlWriter.Flush();
        byte[] soapBuffer = soapStream.ToArray();
        xmlWriter.Close();

        return soapBuffer;
}
```

Event Subscriptions

DPWS supports event subscription under the WS-Eventing specification, allowing other DPWS consumers to register with your device to receive notifications when events of interest to them occur. Subscriptions are handled automatically by the .NET Micro Framework DPWS device stack, so the stack needs to know which events each service provides. The service class declares the events it can provide as part of its constructor, as follows:

```
// Add event source
DpwsWseEventSource sampleEvent = new DpwsWseEventSource("smpl",
      "http://schemas.example.org/SampleService", "SampleEvent");
EventSources.Add(sampleEvent);
```

To send an event, use the `Dpws.Device.Services.DpwsWseSubscriptionMgr.FireEvent` method. This method requires a reference to the `DpwsHostedService`, a reference to the `DpwsWseEventSource`, and a byte array containing the SOAP message to be sent. The latter is typically constructed by a method in the service, using `System.Ext.Xml.XmlWriter`.

The following is an example of raising an event `SampleEvent` in the service `SampleService`, assuming references to the service and the event source were stored in the variables `sampleService` and `sampleEvent` when they were instantiated (as shown earlier):

```
// build the message and fire the event
Dpws.Device.Services.DpwsWseSubscriptionMgr.FireEvent(sampleService, sampleEvent,
      sampleService.BuildSampleEventMessage());
```

Exceptions

The DPWS device stack is built to catch and handle most of the exceptions that can happen during the processing of Web Services messages. Most methods in the processing chain catch the exception and either return a fault message or return null. Null return values are handled in the transport services, where they are converted to exception faults and sent to the listening client. A debug method is also produced. Exceptions will never cause the stack to stop operating.

Faults

The DPWS device stack returns `Addressing` and `Eventing` faults where appropriate, and `Exception` faults for everything else. If one of your operations needs to return a fault, you can do so using the classes in the `Ws.Services.Faults` namespace. The `RaiseFault` method takes one of three enumerations as its second parameter, allowing you to create faults of any of the following types.

- `WsExceptionFaultType`: Exception fault
- `WsWsaFaultType`: Addressing fault
- `WsWseFaultType`: Eventing fault

Threads

The DPWS device stack operates on three main threads, which are started when `Dpws.Device.Device.Start` is called. One is used for processing UDP requests for device discovery, a thread that is not exposed in the APIs. A second is used for the eventing queue and a third for processing HTTP requests. The HTTP thread spawns at most one processing thread for each type of service hosted on the device. These threads last only as long as required to process each HTTP request.

Operations within a particular service are processed synchronously, so only one operation per service can be processed at a time. This is usually acceptable for the applications for which DPWS is designed. If you need to be able to process multiple operations in a particular service simultaneously, you can create multiple services of the same type as long as they have different endpoints.

Using the DPWS Client

A DPWS client derives from `Dpws.Client.DpwsClient` class, which provides a `DpwsDiscoveryClient` member for sending `Probe` and `Resolve` messages. A `Probe` message is a WS-Discovery message used by a client to search for services on the network by service type, and a `Resolve` message is a WS-Discovery message used by a client to search for services on the network by name. The `Probe` and `Resolve` methods of `DpwsDiscoveryClient` both return collections of `DpwsServiceDescription` objects that describe the discovered service endpoints. Once the client has a service endpoint's information, the client can send a request to a Web service.

Sending a Request in the DPWS Client

To make a Web service request from the client, build the XML for the SOAP request using `XmlWriter` and send the request using a `WsHttpClient` object. If the request is a two-way request that returns a response, you will also have to parse the response. The following code example demonstrates how to call a two-way web service method.

```
/// <summary>
/// Method calls a two-way method that sums two integers.
/// </summary>
/// <param name="x">A integer containing the x value to add.</param>
/// <param name="y">A integer containinaining the y value to add.</param>
/// <returns>An integer sum of x+y.</returns>

public int Request(int x, int y)
{
       ...
       // call your function to build the request
       byte[] Request = BuildTwoWayRequest(x, y, sEndPointURI);
       DpwsHttpClient httpClient = new DpwsHttpClient();

       // send the request
       DpwsSoapResponse response = httpClient.SendRequest(Request, sEndPointURI,
              false, false);
       if (response != null)
       {
              // call your function to parse the request
              return Parse2WayResponse(response.Header, response.Envelope);
       }
              else
       {
              ...
       }
}
```

Building a Request

The following code example shows how to create a Web service request.

```
/// <summary>
/// Method builds an Xml 2way request message.
/// </summary>
/// <param name="X">An integer containing the first integer to add.</param>
/// <param name="Y">An integer containing the second integer to add.</param>
/// <param name="endpointAddress">The service endpoint address.</param>
/// <returns>The constructed request.</returns>

private byte[] BuildTwoWayRequest(int X, int Y, string endpointAddress)
{
       MemoryStream soapStream = new MemoryStream();
       XmlWriter xmlWriter = XmlWriter.Create(soapStream);

       // Write processing instructions and root element
       xmlWriter.WriteProcessingInstruction("xml",
              "version='1.0' encoding='UTF-8'");
       xmlWriter.WriteStartElement("soap", "Envelope",
              "http://www.w3.org/2003/05/soap-envelope");

       // Write namespaces
       xmlWriter.WriteAttributeString("xmlns", "wsdp", null,
              Namespaces.GetNamespace("wsdp"));
       xmlWriter.WriteAttributeString("xmlns", "wsd", null,
              Namespaces.GetNamespace("wsd"));
       xmlWriter.WriteAttributeString("xmlns", "wsa", null,
              Namespaces.GetNamespace("wsa"));
       xmlWriter.WriteAttributeString("xmlns", "smpl", null,
              "http://schemas.example.org/SimpleService");

       // Write header
       xmlWriter.WriteStartElement("soap", "Header", null);
       xmlWriter.WriteStartElement("wsa", "To", null);
       xmlWriter.WriteString(endpointAddress);
       xmlWriter.WriteEndElement(); // End To
       xmlWriter.WriteStartElement("wsa", "Action", null);
       xmlWriter.WriteString
              ("http://schemas.example.org/SimpleService/TwoWayRequest");
       xmlWriter.WriteEndElement(); // End Action
       xmlWriter.WriteStartElement("wsa", "From", null);
```

```
        xmlWriter.WriteStartElement("wsa", "Address", null);
        xmlWriter.WriteString(EndpointAddress);
        xmlWriter.WriteEndElement(); // End Address
        xmlWriter.WriteEndElement(); // End From
        xmlWriter.WriteStartElement("wsa", "MessageID", null);
        xmlWriter.WriteString("urn:uuid:" + Guid.NewGuid());
        xmlWriter.WriteEndElement(); // End MessageID
        xmlWriter.WriteEndElement(); // End Header

        // write body
        xmlWriter.WriteStartElement("soap", "Body", null);
        xmlWriter.WriteStartElement("smpl", "TwoWayRequest", null);
        xmlWriter.WriteStartElement("smpl", "X", null);
        xmlWriter.WriteString(X.ToString());
        xmlWriter.WriteEndElement(); // End X
        xmlWriter.WriteStartElement("smpl", "Y", null);
        xmlWriter.WriteString(X.ToString());
        xmlWriter.WriteEndElement(); // End Y
        xmlWriter.WriteEndElement(); // End TwoWayRequest
        xmlWriter.WriteEndElement(); // End Body

        xmlWriter.WriteEndElement();

        // Create return buffer and close writer
        xmlWriter.Flush();
        byte[] soapBuffer = soapStream.ToArray();
        xmlWriter.Close();

        return soapBuffer;
}
```

Parsing a Response

The following code shows how to parse a response.

```
/// <summary>
/// Parses the 2way message response and returns the results.
/// </summary>
/// <param name="header">WsdWsaHeader object: a SOAP response header.</param>
/// <param name="envelope">WsdXmlDocument object: the entire SOAP response.</param>
/// <returns></returns>

private int Parse2WayResponse(WsWsaHeader header, WsXmlDocument envelope)
{
       WsXmlNode tempNode;

       // There should be more validation here; this is the minimal effort required
       if ((tempNode = envelope.SelectSingleNode("Body/TwoWayResponse/Sum", false))
              == null)
       {
              Debug.Print("");
              Debug.Print("Body/TwoWayResponse/Sum element is missing. Returning 0.");
              return 0;
       }
       int Sum = Convert.ToInt32(tempNode.Value);

       return Sum;
}
```

Conclusion

Devices Profile for Web Services (DPWS) is a set of guidelines designed to allow devices to discover each other on a network and invoke the services each provides. The .NET Micro Framework provides a DPWS toolset that allows you to support this profile on your own devices, acting as a server, a client, or both. Microsoft intends to further advance DPWS support in the future. In the meantime, you can model your service code on the samples in this article and provided with.NET Micro Framework v2.5. Prepare to step into the Web Services world and bring a new level of plug-and-play to your networked devices!