Windows Store Device App Design Guide for Specialized Devices Internal to the PC

October, 2013

Abstract

This paper provides information about Windows Store device apps for Windows 8.1 and Windows 8.1 RT. It provides guidance for hardware vendors and OEMs to select a design pattern for building a Windows Store device app for a specialized device internal to the PC. A specialized device internal to the PC is a device that has no user experience built in to the operating system. This paper doesn't provide information about Windows Store device apps for devices that externally connect to Windows over a bus, such as USB, or wirelessly through Wi-Fi or Bluetooth. This information applies to the following operating systems:   
 Windows 8.1

Windows 8.1 RT

References and resources discussed here are listed at the end of this paper.

**Disclaimer**: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet website references, may change without notice. Some information relates to pre-released product which may be substantially modified before it’s commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2013 Microsoft. All rights reserved.



Contents

[Introduction 4](#_Toc370376479)

[Benefits of a Windows Store device app 5](#_Toc370376480)

[Windows Store apps 6](#_Toc370376481)

[Important Windows 8.1 Concepts to Consider For Your App 7](#_Toc370376482)

[Communicating with PC Internal Specialized Devices 7](#_Toc370376483)

[Common elements of a Windows Store device app for a PC internal specialized device 8](#_Toc370376484)

[Custom Driver Access Model 8](#_Toc370376485)

[Accessing PC Internal Devices using In-box Methods 9](#_Toc370376486)

[The device 10](#_Toc370376487)

[The Windows Store device app 10](#_Toc370376488)

[The device metadata package 10](#_Toc370376489)

[The device driver 10](#_Toc370376490)

[Internal and External Specialized Devices 10](#_Toc370376491)

[Acquisition and Installation 11](#_Toc370376492)

[Windows Store Download 11](#_Toc370376493)

[User's View 11](#_Toc370376494)

[Developer's View 12](#_Toc370376495)

[OEM Preload 12](#_Toc370376496)

[User's View 12](#_Toc370376497)

[Developer's View 12](#_Toc370376498)

[Planning and Design 12](#_Toc370376499)

[Device 13](#_Toc370376500)

[Device Identifiers 13](#_Toc370376501)

[Identifiers for Device Metadata 13](#_Toc370376502)

[App 14](#_Toc370376503)

[User Experience 14](#_Toc370376504)

[App Capabilities 16](#_Toc370376505)

[App and Store Manifests 16](#_Toc370376506)

[Privileged Apps 16](#_Toc370376507)

[APIs 17](#_Toc370376508)

[Device Metadata Package 18](#_Toc370376509)

[Apps 18](#_Toc370376510)

[Device Driver 19](#_Toc370376511)

[Driver Packages 19](#_Toc370376512)

[Communicating with Drivers 20](#_Toc370376513)

[Device Interfaces 20](#_Toc370376514)

[Restricted Interfaces 20](#_Toc370376515)

[Windows Runtime device component 21](#_Toc370376516)

[Windows Store Device App development lifecycle 21](#_Toc370376517)

[Publishing Apps for Windows Store Download 21](#_Toc370376518)

[Preinstalling Apps on OEM Systems for PC Internal devices 22](#_Toc370376519)

[Submission Sequence for OEMs and Component Suppliers for OEM preload 23](#_Toc370376520)

[Workflow Case 1: Component supplier develops and distributes 23](#_Toc370376521)

[Workflow Case #2: OEM develops and distributes 24](#_Toc370376522)

[Publishing Device Drivers 26](#_Toc370376523)

[Updating a Windows Store Device App for a PC internal specialized device 27](#_Toc370376524)

[Design Patterns for Specialized Devices 28](#_Toc370376525)

[Summary of PC Internal Specialized Device Design Patterns 28](#_Toc370376526)

[Pattern: WPD Custom Driver 29](#_Toc370376527)

[App 30](#_Toc370376528)

[Finding the Device 30](#_Toc370376529)

[Authoring 30](#_Toc370376530)

[Submission 31](#_Toc370376531)

[Updating 31](#_Toc370376532)

[Pattern: Custom WDF Driver 33](#_Toc370376533)

[Samples 34](#_Toc370376534)

[Planning and Design 34](#_Toc370376535)

[Development Lifecycle 36](#_Toc370376536)

[Appendix A: Custom Driver Access Design Guidance 39](#_Toc370376537)

[Authoring a Device driver 39](#_Toc370376538)

[Communicating Between App and Driver 39](#_Toc370376539)

[Coordinating App and Driver Updates 39](#_Toc370376540)

[Submitting Updates 40](#_Toc370376541)

[Appendix B: Porting existing desktop device apps to Windows Store device apps 41](#_Toc370376542)

[Appendix C: Number of Apps Policy 42](#_Toc370376543)

[Glossary 43](#_Toc370376544)

# Introduction

The Windows device ecosystem is open and diverse. Windows users enjoy access to a plethora of hardware today. People rarely leave their homes without taking their mobile phone, digital camera, or e-book reader. In addition, there's a broad spectrum of devices with unique functionality that addresses specific consumer needs, like glucose monitors, pedometers, barcode scanners, lighting controls, and microscopes. Typically, these devices are external to the PC and connect to a standard external port, such as USB. But they can also be built into the PC.

The broad range of devices available and a rapid innovation cycle present a unique challenge for our hardware partners to provide experiences for new and evolving devices. For several common device types like printers and cameras, Windows 8.1 and Windows 8.1 RT provide built-in experiences that include Windows Runtime APIs for interacting with the device and built-in user experiences that partners can extend.

Windows 8.1 also introduces new Windows Runtime device protocol APIs that Windows Store apps can use to access USB, Human Interface Devices (HID), Bluetooth GATT, Bluetooth RFCOMM, Wi-Fi Direct devices, and more. For more info, see [Windows 8.1: New APIs and features](http://go.microsoft.com/fwlink/?LinkID=298951).

But not every device can be covered by a built-in experience in Windows 8.1. A device that doesn't have an interaction experience built in is a *specialized device*.

Even when there's no built-in user experience, Windows enables you to create a user experience for your PC-internal specialized device. Whatever your internal device may be, you can showcase its unique functionality to users by creating a *Windows Store device app* for your Windows 8.1 or Windows 8.1 RT PC.

Windows Store apps are the focal point of the Windows 8.1 or Windows 8.1 RT PC experience. They are immersive, meaning that they fill the entire screen so there are no distractions for the user — the app developer has full control over every pixel on the screen. Windows Store apps can adapt to a variety of form factors and screen resolutions, such as thin tablets or large desktop monitors, and can work on x86, x64, and ARM platforms.

You can write several Windows Store device apps to expose the rich functionality of your internal devices to users on their PCs. These apps make the PC easier and more enjoyable to use by doing such things as sending data to the web or connecting to a social network.

**IMPORTANT**

* This paper is scoped to Windows Store device apps for Windows 8.1 and Windows 8.1 RT PCs and the specialized devices and hardware therein. These internal devices are likely preinstalled to new PCs before they're shipped to end users.
* This paper doesn't cover devices that the user manually connects to the PC, such as over USB, or wirelessly pairs with the PC, such as over Wi-Fi or Bluetooth.

A Windows Store device app for the PC that uses an internal specialized device is one of many types of Windows Store device apps permitted for the PC. Two other types of Windows Store device apps for specific hardware within the PC might also be of interest to OEMs:

* **Webcam.** See [Windows Store device apps for cameras](http://go.microsoft.com/fwlink/p/?LinkId=306683).
* **Mobile broadband.** See [Mobile Broadband: Enabling Windows Store Mobile Operator Experiences](http://msdn.microsoft.com/library/windows/hardware/hh852368.aspx) in the Windows Dev Center.

There are many differences between Windows Store device apps and desktop apps for the PC and its internal specialized devices. A critical difference that OEMs must understand is in the process of deploying and updating the software. Whereas with a desktop app, an OEM could allow a component supplier to update software independent of the OEM, the update process of a Windows Store device app relies on the OEM to update software components.

## Benefits of a Windows Store device app

A Windows Store device app for the PC complements your specialized device. You can specify a Windows Store device app that's preinstalled on new OEM PCs. You can even design the Windows Store device app to provide a unique experience for each country or region.

Windows Store device apps provide manufacturers of specialized devices an opportunity to leverage their existing Windows investments, sell to a large install base of exciting new PC form factors, and increase post-sale customer engagement.

A well-designed Windows Store device app supports your business model by enabling your customers to easily access services that complement the device experience. Your app can also provide opportunities to offer additional services, accessories, and offer low-cost support.

Windows Store device app benefits include:

* **Discoverability.** Your app can appear on the Start screen when you ship your system. With an app to highlight it, the user can discover the unique functionality your PC offers. Windows 8.1 users also can learn about your app in the Windows Store.
* **Flexibility.** Your app is fully customizable — you control all of the pixels. However, Store policies apply. Be sure your app provides creative, unique value or utility. Don't build a Control Panel for your device.
* **Ease of development.** You can use familiar languages like C++, JavaScript, Visual Basic, and C#.
* **Web integration.** Your app can easily integrate with web services. It's simple for a Windows Store device app to leverage existing web services that are critical to your business.
* **Low-cost support.** Use your Windows Store device apps to direct users to your company's online self-service support.
* **In-app selling opportunities.** Your Windows Store app is a platform from which you can sell upgraded online services or accessories and educate customers about complementary products.
* **Windows integration.** Easily integrate with built-in Windows functionality using app contracts, like the Share contract to share content across apps.

## Windows Store apps

Windows 8.1 introduces a new Windows user interface, which embodies simplicity, gives control to the user, and displays information that's important to them. The interface is a personalized layout with clean typography and animations to make interacting with the PC fluid and intuitive.

The preceding scenarios represent the design target for Windows Store device apps for PC internal specialized devices on Windows 8.1.

A few concepts about Windows Store apps are important to understand before you consider designing an app for your device-specific scenarios. A Windows Store app is an app that runs in the new user experience. Windows 8.1 introduced a new app model for developing apps. [Getting Started with Windows Store app Development](http://msdn.microsoft.com/library/windows/apps/br211386.aspx) is the best reference to quickly learn about this new app model.

Watch [8 traits of great Windows Store apps](http://channel9.msdn.com/events/BUILD/BUILD2011/BPS-1004) and [Platform for Windows Store apps](http://channel9.msdn.com/events/BUILD/BUILD2011/BPS-1005) to learn about the principles and philosophy behind Windows Store apps.

Re-imagine what your app should be like for the Windows Store environment — in a fashion that's consistent with the philosophy and principles of Windows Store apps.

There are fundamental differences between Windows Store apps and desktop (Windows API) apps that are important for hardware partners to understand. See [Getting started with Windows Store apps](http://msdn.microsoft.com/library/windows/apps/br211386.aspx) for more info.

Carefully consider if your app scenarios are better suited to Windows Store apps or the desktop environment. For example, Windows Store apps are per-user, so they aren't appropriate for system-wide configuration settings. A Windows Store device app may support configuration of a personal device (such as a media player) if the device is for a particular user and only the settings on the device itself are being changed. As you re-imagine your app and consider your scenarios, research whether there are Windows Runtime APIs to complete your scenarios. See Appendix B for considerations for porting desktop apps from Windows 7.

#### Control Panel or settings-only apps

Here are a couple of important considerations for those who perceive that they need a Control Panel or settings-only Windows Store app:

* Windows Store apps are per-user apps. It isn't appropriate for a Windows Store app to modify system settings. It's okay for an app to change settings on a device that only it can talk to, but it may not modify Windows system-wide.
* Your app should contain your own unique visuals. Don't mimic the visual style of the Windows Settings app or a Control Panel app. Users should not confuse your app for Microsoft provided Windows UI.

It's up to the OEM to carefully choose the apps that best showcase the functionality of their PC and are consistent with the design guidelines for Windows Store apps. Not every app for a device inside of the PC is a good candidate for a Windows Store app, and the OEM needs to decide which apps add the most customer value for each of their PCs.

### Important Windows 8.1 Concepts to Consider For Your App

The great thing about a Windows Store device app is that it's an app – and it can do all of the things Windows Store apps can do. Here are a few key concepts that apply to all Windows Store apps, with some examples that show why they may apply to an app for your device.

#### Contracts

Contracts allow apps to work together, making it easy to search, share, and send content between them. Windows Store apps can communicate with one another in Windows 8.1. For example, a Windows Store device app for a phone can use the Share contract to send photos to an app for Facebook or Flickr.

See [App Contracts](#_App_Contracts) in this document for more information.

#### Live tiles

Live tiles help your apps come alive with activity and show the user the latest content so they can stay up-to-date at a glance. For example, a Windows Store device app for a phone can use live tiles to display recent photos from the phone.

To learn more about live tiles, see [Make great Windows Store apps: Invest in a great tile](http://msdn.microsoft.com/library/windows/apps/hh464920.aspx#traits_5_great_tile) and [Guidelines for tiles](http://msdn.microsoft.com/en-US/library/windows/apps/hh465403.aspx).

#### Toast notifications

Toasts are notifications that you show to users when your app is in the background. They're great at updating users with information they want to know immediately. Users tap the toast to switch to your app and learn more.

To learn more, see [Make great Windows Store apps: Stay connected and alive](http://msdn.microsoft.com/library/windows/apps/hh464920.aspx#traits_6_conntected).

# Communicating with PC Internal Specialized Devices

The following table is a simple grouping of design patterns that Windows Store device apps can use to communicate with a PC internal specialized device. A design pattern represents the programming model that enables an app to access a specialized device and transfer information to and from it.

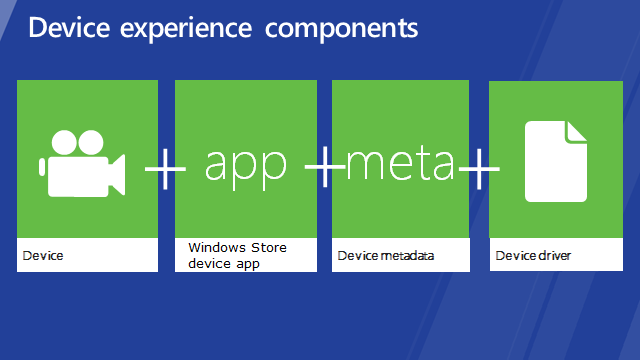
Although it's only one facet of building an app that completes your scenarios, device access is an essential part of building a Windows Store device app.

|  |  |  |
| --- | --- | --- |
| Design pattern | Connectivity | Description |
| Portable Device (WPD) | Internal | Device that transfers objects with properties and methods. |
| Device Protocol APIs | Internal | Using the USB and HID APIs to communicate with PC internal devices. |
| Custom Driver Access | Internal | Transfers commands and data using device IO control. |

**IMPORTANT**   Device access using custom drivers requires approval from Microsoft. OEMs and IHVs that want to implement device access for a specialized device using a custom driver should first contact their Microsoft contact to discuss their scenario with the Windows Ecosystem team. For more information see the Custom Driver Access Model section.

# Common elements of a Windows Store device app for a PC internal specialized device

The Windows Store device app you provide to your customers depends on four basic components. Depending on your particular device you may not be required to provide all of these components yourself. The components required are determined by the Design Pattern associated with the device.



## Custom Driver Access Model

Custom driver access in Windows 8.1 and Windows 8.1 RT is a limited use device access model to enable rapid device innovation and experiences for new and evolving devices. The following are important considerations when using the Custom Driver Access Model.

* Custom driver access should be rarely used and constrained and is not meant to replace in-box capabilities in Windows 8.1 and Windows 8.1 RT – if there is an in-box method to accomplish a device scenario, that in-box support must be used over a custom driver. Windows 8.1 provides several in-box methods to communicate with devices. For more information see, [Devices: new or updated in Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/apps/bg182882.aspx).
* Custom Driver Access is available only to OEMs - IHVs and component suppliers must work through their OEM to enable custom driver access to their devices.
* Custom drivers must first be approved by Microsoft before they can either be pre-installed or downloaded from Windows Update.
* Any Windows Store app that intends to access a restricted interface using a custom driver must be approved for access from that Windows Store app by Microsoft – this includes apps installed on Windows 8.1 and Windows 8.1 RT PCs.
* Approval of using a custom driver does not imply automatic approval for accessing the custom driver interfaces with a Windows Store Device app. Windows Store device app access to a custom driver also requires approval and discussion with Windows Ecosystem team.

Details of using custom drivers are discussed in more detail in the sections that follow. OEMs with custom driver scenarios should first contact their Microsoft contact to discuss their scenario with the Windows Ecosystem team.

## Accessing PC Internal Devices using In-box Methods

Windows 8.1 introduces new Windows Runtime device protocol APIs that Windows Store apps can use to access USB, Human Interface Devices (HID), Bluetooth GATT, Bluetooth RFCOMM, Wi-Fi Direct devices, and more. For more info, see [Windows 8.1: New APIs and features](http://go.microsoft.com/fwlink/?LinkID=298951).

Starting in Windows 8.1, in-box support is provided for the following classes of devices in addition to the onboard devices supported in Windows 8 like webcams, sensors and audio.

* Human Interface (HID)
* Point of Service (PoS)
* Universal Serial Bus (USB)
* Bluetooth
* 3D Printers

OEMs can access PC internal devices connected to the PCs USB bus by providing a Windows Store app privileged access to the Device Container representing the PC. Accessing PC Internal devices using one of these methods is preferred to creating a custom driver and does not require approval from Microsoft. Only privileged apps designed by the OEM for the Device Container may access PC internal devices using the USB and HID APIs. For more information, see [Building Windows Store device apps for Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265148(v=vs.85).aspx).

## The device

The first element is the PC internal specialized device. For a PC Internal specialized device, the physical device — the "piece of plastic" — that the user interacts with is the PC itself.

When the user sets up a new PC, Windows identifies the manufacturer and model of the PC.

## The Windows Store device app

The second element is the Windows Store device app. The app provides a customized user experience for the specialized device, enabling the user to access the device's unique features.

## The device metadata package

The third element is the device metadata package. This is an extended version of any device metadata package you may have already created for Windows 7. In Windows 8.1, device metadata creates the link between PC (and its internal devices) and its Windows Store device app. In addition to UI content for the PC (localizable model name, description, and photorealistic icons) the device metadata package indicates which app has privilege to access the device. Windows automatically downloads device metadata from the Windows Metadata Internet Service (WMIS).

For PC internal specialized devices, device metadata is bound to the Device Container representing the PC. Device metadata for the PC is managed by the OEM and can also be preinstalled as part of a new PC. For more info, see [Specifying Hardware IDs for a Computer](http://msdn.microsoft.com/en-us/library/windows/hardware/ff552325(v=vs.85).aspx) and [Device Metadata Package Schema Reference](http://msdn.microsoft.com/en-us/library/windows/hardware/Ff541452(v=vs.85).aspx).

## The device driver

The final element is the device driver. Drivers are extensions to Windows that allow apps to communicate with devices. Windows includes drivers for most common devices, as well as drivers that enable communication to many specialized devices. Windows automatically searches the in-box drivers and Windows Update when it discovers a new device. If no driver that supports your device is included or found, you must create your own.

**IMPORTANT**   Device access using custom drivers requires approval from Microsoft. OEMs and IHVs that want to implement device access for a specialized device using a custom driver should first contact their Microsoft contact to discuss their scenario with the Windows Ecosystem team. For more information see the Custom Driver Access Model section.

# Internal and External Specialized Devices

Windows supports Windows Store device apps for devices externally connected to the PC, devices on a local network, and devices built into the PC. While they may seem very different to the end user, the ways you construct the apps are very similar. The differences lie in who controls which apps are available to the end user and how they are distributed.

A network-attached device is a device that's connected to the PC's local network, whether wired or wirelessly. Networked devices are considered external devices.

An external device is any device attached to an external port on the PC, like a USB port, or inserted into an external expansion slot, like an SD or ExpressCard slot. The external device could also communicate with the PC using wireless technology like Bluetooth. For an external device, which apps are automatically downloaded and which have access to the device is determined by the device manufacturer.

An internal device is any device built into the PC, or inserted into an expansion slot inside of the PC, like a PCI-e or mini-PCI slot. For internal devices, which apps have access to the device is determined by the PC manufacturer. This is true both for original equipment and for aftermarket add-on components.

The PC internal specialized device scenario works best for devices that are preinstalled and ship as part of a new PC. It can work with retail/aftermarket devices, like a graphics card or PC internal expansion cards that are added by the user afterwards ‑ as long as all software components are preinstalled with the PC and are carefully managed by the OEM. Because there's no single OEM managing the device metadata for all supported components, the PC internal specialized device scenario isn't designed for a scenario supporting an enthusiast who builds their own PC from scratch, starting with a motherboard.

# Acquisition and Installation

A PC internal specialized device supports these acquisition scenarios, compared to external devices.

|  |  |  |
| --- | --- | --- |
| Supported acquisition scenarios | External device | Internal device |
| Automatically acquired (automatic installation) |  |  |
| Windows Store download  (manual installation) |  |  |
| OEM preload  ([preinstallation using DISM](http://go.microsoft.com/fwlink/p/?LinkId=325524)) |  |  |

**IMPORTANT**Automatic acquisition of a Windows Store app is not supported by internal devices in the PC. Windows will only automatically acquire a Store app for externally connected peripheral devices. For more information on building Store apps for externally connected devices, see [Building Windows Store device apps for Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265148(v=vs.85).aspx). For more information about automatic acquisition of Store apps, see [Automatic installation for Windows Store device apps in Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265137(v=vs.85).aspx).

The following app acquisition scenarios that apply to PC internal specialized devices that follow are described from both a user's and a developer's point of view.

## Windows Store Download

### User's View

1. The user goes to the Windows Store to search for an app by name, like "Contoso DVR", and acquires it.
2. The app installs from the Store, just like any other app in the Store.

### Developer's View

1. User signs in to the Windows Store with a Microsoft account.
2. User goes to the Windows Store, then finds and acquires the app.
3. The app is downloaded from the Windows Store. The app tile appears in the user's Apps list.

PC Internal specialized devices can only be accessed by a privileged app. The privileged app can't access the device until device metadata for the PC has been acquired by Windows. Details follow on this, the primary reason we recommend that OEMs preinstall device metadata for the PC.

Software ‑ especially the app and any drivers ‑ should be designed to cope gracefully if any companion components are unavailable.

## OEM Preload

OEM Preload is recommended for a PC internal specialized device. For more information about OEM preload, see [Preinstall Apps Using DISM](http://go.microsoft.com/fwlink/p/?LinkId=325524).

### User's View

1. User gets a new OEM PC, turns it on, and signs in to it.
2. They see the device app on the Start screen.

### Developer's View

Device drivers, device metadata, and app should be preinstalled by an OEM using the Windows Assessment and Deployment Kit (ADK).

1. During initial power on by an end-user, any device drivers that match present hardware are installed.
2. Next, the Windows out-of-box experience prompts the user to opt in to default settings and completes.
3. User logs in to Windows for the first time on that machine, and Device Metadata is installed once for the entire machine.
4. The Windows Store device app is installed for that User who sees the app on their Start Screen, and the app is installed for each subsequent user when they first log into Windows.

Software ‑ especially the app and any drivers ‑ should be designed to cope gracefully if any companion components are unavailable.

# Planning and Design

A good Windows Store device app depends on the three other components described previously: the device, the device metadata, and the device driver. This section outlines:

* How Windows identifies your device.
* How Windows matches device metadata to your device.
* What your driver must do to enable communication with the app.
* Recommendations on how to tie your device in to the power of a Windows Store app.

If you already have a device that works with Windows, you may be familiar with some or all of these components. Even so, this section identifies the important points about how all these components tie together.

## Device

As described previously, a device can be just about any hardware the PC can communicate with. For the purposes of a Windows Store device app for a PC Internal specialized device, the scenario works best when the device and all necessary software are preinstalled with a new PC.

**Note:** Windows Store device apps can only access Plug and Play (PnP) devices. Access to devices requires a device interface that's only possible with PnP devices.

### Device Identifiers

To find the appropriate software, Windows relies on the device to provide one or more *device IDs.* Devices can report multiple IDs with different levels of detail. Windows supports two types of device IDs for the purposes of matching drivers to devices:

* **Hardware IDs** describe the device in terms of make/model for the purposes of selecting device drivers to control the device.
* **Compatible IDs** describe the device in terms of standards that it complies with, such as the USB mass-storage device class

For example, a USB mouse has these IDs:

Hardware IDs:

* USB\VID\_045E&PID\_0047&REV\_0300
* USB\VID\_045E&PID\_0047

Compatible IDs:

* USB\Class\_03&SubClass\_01&Prot\_02
* USB\Class\_03&SubClass\_01
* USB\Class\_03

To find the right software, Windows uses the hardware and compatible IDs to search Windows Update, Windows Metadata Internet Services (WMIS) and the Windows Store to find drivers, metadata, and apps that go with the device.

## Identifiers for Device Metadata

#### PC Internal specialized devices

Model IDs don't apply where the device is the PC, which matters to all PC internal specialized devices that depend upon device metadata for privileged app declarations. Instead, a hardware ID for the PC is used to associate device metadata with the PC. Throughout the remainder of this doc, we refer to the *Hardware ID for the PC* as the identifier used to associate device metadata with the PC. For more information about how device metadata for the PC container is bound, see [Specifying Hardware IDs for a Computer](http://msdn.microsoft.com/en-us/library/windows/hardware/ff552325(v=vs.85).aspx).

Use the ComputerHardwareIDs.exe tool in the Windows Driver Kit (WDK) to generate the correct IDs, and carefully determine, based upon reading above document section, which of the generated IDs is best to put in your device metadata for the PC.

#### Choosing Device Identifiers for your Device

Device Identifiers play a very important role in matching the components of your Windows Store device app to your device. The IDs are used for acquiring different pieces of software:

|  |  |  |
| --- | --- | --- |
| Device ID Type | Acquire Device Drivers | Acquire Device Metadata |
| Hardware ID | Yes | Yes |
| Compatible ID | No | No |
| Model ID | No | Not for PC Internal specialized devices |

It's important that you design your device to provide appropriate IDs so that you can uniquely select your drivers and device metadata. For example, if you use a commodity part to provide the USB interface for your device, make sure that your device provides Vendor and Product IDs that uniquely identify it. Don't use default IDs provided by the interface controller or IDs from the vendor's sample code.

Without the proper IDs, you can't target your drivers and your app to your device.

## App

An app provides the customer with a PC experience to match your device's unique capabilities.

### User Experience

Essentially, your Windows Store device app provides a custom user experience with your device. The user sees this app as an extension of your device as part of the PC. A great app is able to enhance your device and delight your customers.

Because of this, the design of your Windows Store device app and the experience it provides should be considered as seriously as the design of the device itself. An app that only enables tweaking esoteric configuration parameters of your device, or that just displays brand information, has little value to average users. An app that integrates well with the look and feel of Windows, that provides a unique way to work with your device, and that extends your device to other Windows Store apps is very valuable to your customers.

The following are some recommendations and best practices on how your app can integrate your device into Windows.

#### App Contracts

Windows Store apps interact with each other only through the [Windows app contracts](http://msdn.microsoft.com/library/windows/apps/hh464906.aspx). These are formal interfaces that allow apps to invoke other apps or share information with them. Just as your Windows Store device app is an extension of your device as part of the PC, consider the contracts from the point of view of invoking features on or sharing information with your device.

The following table briefly describes the contracts most relevant to Windows Store device apps. It also provides examples of how your app might use it to enable access to your device. However, this isn't a complete list of the contracts and extensions available to Windows Store device apps, and your app is free to use any other provided capabilities as well.

|  |  |  |
| --- | --- | --- |
| **Contract** | **Description** | **Example Uses** |
| Search | When you participate in this contract, you agree to make your app's content searchable by other participants. | Allow the user to search the content on your device from the Search charm. |
| Share | Apps that support the sharing contract can automatically share content to and from any other app that also supports the contract. | As the source, you can share content from your device with another app, like sending content from your device through another app to a social media site.  As the target, you can receive content from another app to store on your device, like pushing a photo from a social media app onto your device. |
| App-to-App Picking | You can help users pick files from one app directly within another app. Users gain freedom and flexibility to choose files that are stored and presented by apps. | As the target, you can allow other apps to access data on your device, allow other apps to save data to your device, and notify other apps when the contents of files from your device have been changed. |
| Settings | You can provide quick, in-context access to settings that affect the user's experience with your app. | Rather than clutter your app, you can allow users to configure advanced device settings or invoke customized Help through the Settings charm. |

#### Handling Device Changes and Errors

Windows users are accustomed to connecting and disconnecting their devices, and your Windows Store device app should be able to gracefully handle conditions like:

* The user launches the app before connecting the device.
* The user unplugs the device in the middle of an operation, or while your app is in the background and suspended.
* The user has multiple of your device connected to the PC at the same time and needs to be able to choose which one to interact with.
* The user has your device connected, but for some reason attempts to access it fail.

See the Windows.Devices.Enumeration namespace for further information about how to detect some of the listed conditions.

### App Capabilities

Windows requires that Windows Store apps list what capabilities they access so that users are aware of and have control over app capabilities. The capabilities control whether the app can access such things as the Internet, a private network, the user's libraries (documents, pictures, and videos), the user's devices (location, microphone, and so on).

The most relevant capability for an app that accesses a PC internal specialized device is the Custom device capability.

The app must declare these capabilities in its app manifest. If the capability isn't asserted (or has been rescinded by the user), the app receives an error when it tries to access the associated resource.

These capabilities are not supported in the app manifest editor in Visual Studio®. The developer must add a DeviceCapabilites element to the Capabilities element in package.appxmanifest. See [DeviceCapability](http://msdn.microsoft.com/library/windows/apps/br211430.aspx) in the Windows Dev Center for more information on possible settings.

### App and Store Manifests

The app and store manifests are XML files included in the app package. The app manifest provides information about the app's UI, tiles, [capabilities](#_App_Capabilities), supported contracts, and packaging. Fields from the packaging information are used to identify which Windows Store device apps have privileged access to the device. The fields used are:

|  |  |
| --- | --- |
| Name | Description |
| Package Name | Specifies the unique name that identifies the package on the system |
| Publisher | Specifies the subject field of the signing certificate used to authenticate the package |

The package and publisher names are used by the [device metadata system](#_Device_Metadata_Package) to identify the Windows Store device apps for a given device.

### Privileged Apps

To prevent Windows Store apps that aren't approved by the device vendor from communicating with the device, the vendor can restrict access to a limited number of approved Windows Store device apps.

Apps that are allowed access by the device vendor to a restricted interface are *privileged apps*. Privileged apps are identified in device metadata. Only privileged apps can access a restricted interface. The device software determines what restricted capabilities a privileged app may access on the device, and the app declares which ones it wants to use.

All PC internal specialized devices are considered restricted interfaces and can only be accessed by privileged apps.

### APIs

The Windows.Devices namespace provides classes for discovering devices attached to the system and for accessing many common devices. Your app uses these to find the device and communicate with it.

#### Device Discovery

Most devices can be programmatically discovered using the classes in the Windows.Devices.Enumeration namespace. The DeviceInformation class allows you to query the system for available devices, and you can use the DeviceWatcher class to get notifications when the results of a query change, generally in response to the user connecting or disconnecting a device from the system.

The DeviceInformation class also provides access to UI information for your device, like its display name and the device's icon. For more detailed information about the device, you can query for its related device and device container using the Windows.Devices.Enumeration.Pnp.PnpObject class. For example, the properties specified for the device container in the device metadata package are available through the PnpObject that represents the device container.

See [Quickstart: Enumerating commonly used devices](http://msdn.microsoft.com/en-us/library/windows/apps/hh464974.aspx) for information and guidelines about building your own device selection dialog using Windows.Devices.Enumeration.

#### Device Access

Once your app has found the device(s) it will work with, Windows provides a built-in set of device protocol APIs for various types of devices. The most relevant of these for PC internal specialized devices are Windows.Devices.Portable and the Device Access interfaces for accessing devices with custom drivers. More specific examples are provided in the patterns sections of this document.

Windows 8.1 also introduces new Windows Runtime device protocol APIs that Windows Store apps can use to access USB, Human Interface Devices (HID), Bluetooth GATT, Bluetooth RFCOMM, Wi-Fi Direct devices, and more. For more info, see [Windows 8.1: New APIs and features](http://go.microsoft.com/fwlink/?LinkID=298951).

Starting in Windows 8.1, in-box support is provided for the following classes of devices in addition to the onboard devices supported in Windows 8 like webcams, sensors and audio.

* Human Interface Devices (HID)
* Point of Service (PoS)
* Universal Serial Bus (USB)
* Bluetooth
* 3D Printers

OEMs can access PC internal devices connected to the PCs USB bus by providing a Windows Store app privileged access to the Device Container representing the PC. Accessing PC Internal devices using one of these methods is preferred to creating a custom driver and does not require approval from Microsoft. Only privileged apps designed by the OEM for the Device Container may access PC internal devices using the USB and HID APIs. For more information, see [Building Windows Store device apps for Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265148(v=vs.85).aspx).

## Device Metadata Package

Device metadata defines which Windows Store device app should be downloaded for the device, and which Windows Store device apps have privileged access to the device's interfaces.

By default, Windows provides a very simple view of a device in the Settings app and the Devices and Printers item in Control Panel, using information retrieved from the device itself. Vendors can augment this presentation by creating a device metadata package for the device and submitting it to the [Windows Dev Center hardware dashboard](http://msdn.microsoft.com/en-us/windows/hardware/br257577). Device metadata enables you to provide more detailed ‑ and localizable ‑ information about your device, such as user-friendly model and manufacturer names, photorealistic icons, and descriptive text.

Metadata is usually associated with a device container. When Windows identifies a new container, it searches WMIS for corresponding metadata using either the container's model ID or using the hardware IDs of all of the devices in the container. If matching metadata is found, Windows downloads the package and installs it automatically.

A device metadata package consists of a set of XML files that describe the device, the software associated with the device, and the package itself. These are combined into a.cab file, which is renamed to a .devicemetadata-ms file.

Microsoft provides the Device Metadata Authoring Wizard that can be used to create new packages or adapt existing packages to include the new elements needed to support a Windows Store device app.

After installing the tools, as described in [Getting started with Windows Store device apps](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265140(v=vs.85).aspx), the Device Metadata Authoring Wizard (DeviceMetadataWizard.exe) can be found at: %ProgramFiles(x86)%\Windows Kits\8.1\bin\x86

For more information about how device metadata works, and how to test and debug it, see [Device Metadata Package Pipeline](http://msdn.microsoft.com/en-us/library/windows/hardware/gg463157.aspx).

### Apps

Device metadata can specify which app should be automatically downloaded when an externally connected device is attached. The metadata contains the Package Name, Publisher, and App ID of the app to be downloaded. These values must match for Windows to identify and download the correct app. An app has a Store Manifest that applies to all Windows Store device apps, including those for PC internal specialized devices. See [App and Store Manifests](#_App_and_Store) for more information on how these values are obtained.

The metadata also lists one or more apps that have privileged access to the device. Apps require privilege to access a restricted device, and the device metadata is where that privilege is granted. For each privileged app, the metadata must provide the Package name and Publisher, and must indicate whether that app has access to any Windows Driver Framework (WDF) custom drivers that control the device.

## Device Driver

The device driver allows your app to communicate with your device using the device APIs previously described. Every Windows Store device app solution in this paper makes use of one or more device drivers. Some use drivers included with Windows (*in-box*), others use custom drivers provided by the manufacturer. Solutions that can use an in-box driver are much simpler to create and support, but are not applicable to all devices. The per-pattern sections that follow provide more details for various types of devices and what sort of drivers are required.

**IMPORTANT**

* Device access using custom drivers requires approval from Microsoft. OEMs and IHVs that want to implement device access for a specialized device using a custom driver should first contact their Microsoft contact to discuss their scenario with the Windows Ecosystem team. For more information see the Custom Driver Access Model section.
* Windows 8.1 introduces new Windows Runtime device protocol APIs that Windows Store apps can use to access USB, Human Interface Devices (HID), Bluetooth GATT, Bluetooth RFCOMM, Wi-Fi Direct devices, and more. For more info, see [Windows 8.1: New APIs and features](http://go.microsoft.com/fwlink/?LinkID=298951).
* OEMs can access PC internal devices connected to the PCs USB bus by providing a Windows Store app privileged access to the Device Container representing the PC. Accessing PC Internal devices using one of these methods is preferred to creating a custom driver and does not require approval from Microsoft. Only privileged apps designed by the OEM for the Device Container may access PC internal devices using the USB and HID APIs. For more information, see [Building Windows Store device apps for Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265148(v=vs.85).aspx).

### Driver Packages

Device drivers are distributed and installed in driver packages. This is a signed collection of files that includes the driver binaries, an INF file that tells Windows what devices the package applies to and how to install it, and a signed catalog of the package contents, which is used for integrity checking.

Windows maintains a database of the driver packages that have been installed on the system (the driver store). When a new device is attached, Windows searches this database for the best driver package and automatically installs it. Windows can search Windows Update when no local match is found. If Windows Update provides a driver package it will be automatically added to the driver store and applied.

### Communicating with Drivers

Apps can only talk to devices by going through a device driver. Windows provides a very low-level interface between desktop apps and the device drivers. This interface has three core operations: write a blob of data to the driver, read a blob of data from the driver and perform a driver-specific I/O control operation, or *ioctl*. All of the higher-level Windows API and Windows device services, including the Windows.Devices interfaces, build upon these three basic operations.

### Device Interfaces

A typical device driver supports several driver-specific ioctls to expose the unique capabilities of the device. These ioctls often tunnel through directly to the device. For example, the driver for a floppy disk uses the read/write primitives to transfer data to and from the media and provides a set of custom ioctls to determine the type of media in the drive, format tracks, and so on.

The driver can publish one or more *device interfaces* for the device during initialization. A device interface is a loosely defined software contract indicating that the device driver supports a specific combination of ioctls for that particular device. Each contract has a unique identifier (a GUID) and apps can search for devices using those identifiers. For example, a credit card reader's driver publishes a device interface with the system for each reader it controls, and apps that want to work directly with that reader can search for that GUID to learn which ioctls the driver supports.

Windows defines a number of well-known device interfaces for various types of device drivers. Drivers can also define their own custom interfaces, and app writers can use those interfaces to provide vertical solutions.

### Restricted Interfaces

Microsoft provides some interfaces in Windows 8.1 and Windows 8.1 RT that are identified as restricted interfaces. An example is accessing a custom driver.

By default, the low-level device primitives (read/write/ioctl) aren't available to Windows Store apps. Your custom driver can enable a Windows Store device app to access one of the device interfaces by marking it restricted, listing your Windows Store device app as one of a limited number of privileged apps in the device metadata, and declaring the device interface(s) your app will access in the app's manifest. A restricted interface can still be accessed by a desktop app, but it can't be accessed by any non-privileged app.

All privileged apps for the PC have access to all restricted interfaces exposed by drivers associated with the PC device container that matches their declared device capabilities.

**IMPORTANT**   Device access using custom drivers requires approval from Microsoft. OEMs and IHVs that want to implement device access for a specialized device using a custom driver should first contact their Microsoft contact to discuss their scenario with the Windows Ecosystem team. For more information see the Custom Driver Access Model section.

### Windows Runtime device component

Device drivers, even custom device drivers, that publish one of the standard Windows device interfaces work with the existing Windows infrastructure. For example, a device that reports the interface for Windows Portable Device (WPD) is accessible through the Windows.Devices.Portable APIs.

For an app to use a custom device interface on the device, it must use the Device Access APIs. The driver writer can provide a Windows Runtime component written in C++ that exposes the device's interfaces into higher-level languages like C# or JavaScript. The Windows Store device app then includes the device projection DLL within its app package.

This Windows Runtime device component defines one or more public WinRT classes that represent the capabilities of the device. The classes can make use of all of the high-level WinRT capabilities, such as properties, events, and asynchronous tasks. These classes then implement the corresponding calls to the Device Access APIs to control the device.

The app writer can also communicate with the device by using the Windows.Devices.Custom.CustomDevice.FromIdAsync method to open the device and the SendIOControlAsync method to transfer information to and from the device from higher-level languages like C#.

# Windows Store Device App development lifecycle

The lifecycle of a Windows Store device app project – development, deployment, and updating – are covered in the [Building Windows Store device apps for Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265148(v=vs.85).aspx). Read it to learn the general process and requirements for deploying a Windows Store device app.

Because all Windows Store device apps for PC internal specialized devices depend on privileged apps and aren't automatically acquired, we recommend using a slightly different submission order from that described in the white paper for PC internal specialized devices.

See the per-pattern sections later in this document.

## Publishing Apps for Windows Store Download

A Windows Store device app can be published to the Store for users to download themselves like any other app. Only these three steps of the sequence are required for this acquisition scenario. These steps can be performed in parallel but all three steps must be complete for the Windows Store app to successfully interact with the internal device.

|  |  |
| --- | --- |
| Step | Description |
| 1 | Submit custom device drivers (if any) and wait for distribution. |
| 2 | Submit the app to the Windows Store Dashboard. You do not have to wait for the app to be live in the Windows Store to proceed to the next step but you will need the Package name and Publisher name details of your app to proceed with step 3. Users will not be able to download your app until it has been accepted by the Windows Store. |
| 3 | Submit device metadata package with privileged apps listed.  Device metadata submissions without any validation errors normally take 48 hours to publish. However, Windows only automatically downloads new device metadata for the PC every 14 days as part of your PCs maintenance tasks. You should wait 14 days after your metadata has been published for distribution to PCs. In the interim, users who manually download the app before device metadata has been delivered won't be able to access your device. |

**IMPORTANT**Automatic acquisition of a Windows Store app is not supported by internal devices in the PC. Windows will only automatically acquire a Store app for externally connected peripheral devices. For more information on building Store apps for externally connected devices, see [Building Windows Store device apps for Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265148(v=vs.85).aspx). For more information about automatic acquisition of Store apps, see [Automatic installation for Windows Store device apps in Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265137(v=vs.85).aspx).

## Preinstalling Apps on OEM Systems for PC Internal devices

An OEM may preinstall drivers and a limited number of Windows Store device apps for devices preinstalled in a new PC shipped to customers. These steps can be performed in parallel but all three steps must be complete for the Windows Store app to successfully interact with the internal device.

|  |  |
| --- | --- |
| Step | Description |
| 1 | Submit custom device drivers (if any). |
| 2 | Submit the app to the Windows Store. You do not have to wait for the app to be live in the Windows Store to proceed to the next step but you will need the Package name and Publisher name details of your app to proceed with step 3. Users will not be able to download your app until it has been accepted by the Windows Store. |
| 3 | Submit the one device metadata package for the PC with PC hardware IDs and privileged apps listed.  OEMs can directly download device metadata from the hardware dashboard in the Windows Dev Center (https://sysdev.microsoft.com), under Device Metadata > Manage Experiences. |
| 4 | Preinstall drivers, device metadata, and the app into the image to be deployed to your systems. |

Important aspects of this sequence:

* All preinstalled apps must go through the Store and pass Store Certification.
* Access permissions for PC internal specialized devices are specified in the PC's device metadata. There's no separate metadata submitted by the OEM for individual components. The OEM, not the component suppliers, governs and manages the PC metadata that spans all PC internal components.

Drivers and device metadata should be deployed with the system image so that they can be installed with the operating system for the best user experience ‑ particularly when users aren't connected to the Internet.

## Submission Sequence for OEMs and Component Suppliers for OEM preload

Including the correct versions of all of the components in an OEM preload image is a good way to make sure they're all present for bindings to succeed. Which workflow is used depends on who develops and distributes the app. For more information about OEM preload, see [Preinstall Apps Using DISM](http://go.microsoft.com/fwlink/p/?LinkId=325524).

### Workflow Case 1: Component supplier develops and distributes

In this case, the component supplier owns, develops, and distributes the app and driver that access your PC internal specialized device. The OEM owns the device metadata.

#### Component supplier internal development

* The component supplier develops the app and driver as you would normally for your design pattern. The supplier creates temporary device metadata that binds to the PC Device container you use for internal development and test. For more information about how the device metadata for the PC container is bound, see [Specifying Hardware IDs for a Computer](http://msdn.microsoft.com/en-us/library/windows/hardware/ff552325(v=vs.85).aspx).   
  **Note:** To create device metadata with the Device Metadata Authoring Wizard, select the **Internal device** option. After installing the tools, as described in [Getting started with Windows Store device apps](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265140(v=vs.85).aspx), the Device Metadata Authoring Wizard (DeviceMetadataWizard.exe) can be found at: %ProgramFiles(x86)%\Windows Kits\8.1\bin\
* The supplier develops and tests using that device metadata, but doesn't submit the metadata to Microsoft.
* The supplier hands off the app and device metadata information for the privileged app to the OEM.

#### OEM device metadata development

* The OEM authors device metadata for the PC that's targeted to hardware IDs for PCs that the OEM plans to ship.
* The OEM merges new privileged app entries that the component supplier provided into OEM metadata for the PC, in the [PrivilegedApplications element](http://msdn.microsoft.com/library/windows/hardware/jj159067(v=vs.85).aspx).
* The OEM updates device metadata to bind to the targeted PC model.
* The OEM tests using the merged metadata bound to the PC model with the correct versions of all of the software components.
* The OEM confirms that the privileged app fields in the PrivilegedApplications element and the device metadata match so that the app can access any PC internal specialized devices with restricted interfaces before the OEM submits the device metadata for deployment.

#### Submission

The following steps show the deployment of an app supplied by the component supplier. For the example of a custom IR blaster, the components involved include the device driver, device metadata, and the app developed by a component supplier as an app to be preinstalled on a new tablet:

1. The component supplier passes driver and Hardware Certification tests, including Device.Devfund.CDA
2. The supplier submits the device driver to the Windows Dev Center hardware dashboard for certification and distribution via Windows Update
3. The device driver passes the WACK(Windows App Certification Kit) tests.
4. The supplier submits the app to the Windows Store Dashboard.
5. The OEM submits the device metadata package to the Windows Dev Center hardware dashboard for distribution via Windows Metadata and Internet Services (WMIS). The OEM can directly download the device metadata for preload from the dashboard's Manage Experience page.
6. The OEM can now preload the certified and digitally signed device driver, device metadata, and app.

When submitting a privileged app for a PC internal device for Windows Store download, follow the instructions in [Publishing Apps for Windows Store Download](#_Publishing_Apps_for), but remember that the metadata in this case is the device metadata package for the PC, not for your device, and the OEM needs to submit that device metadata package.

### Workflow Case #2: OEM develops and distributes

In this case, the OEM develops and distributes an app that accesses one or more PC internal specialized devices from different component suppliers. The OEM ultimately owns app development, app distribution, and device metadata maintenance. The component supplier owns the driver. It's up to the two parties to figure out the right relationship for sharing specifications and code and for co-development of such an app. An OEM might take an app developed by a supplier and submit it under their own name.

#### OEM specification

* The OEM specifies and declares what dependencies their app has on the component supplier. For example, any requirements on functional contracts of device interface class, and any UX/presentation layer requirements. An OEM may choose to define a standard Device Interface Class as a functional contract for their component suppliers to follow for a set of specific functionality.
* It's up to the OEM and the component supplier to determine how to share code to achieve their goals.

#### Component supplier internal development

* The component supplier develops a private sample or test app and follows OEM specifications for apps and components:
  + The app meets the Windows App Certification Requirements.
  + Additional OEM requirements for functional contracts, if any, are built into the driver.
* The supplier creates temporary device metadata that binds to the PC Device container used for internal development and test. For more information about how the device metadata for the PC container is bound, see [Specifying Hardware IDs for a Computer](http://msdn.microsoft.com/en-us/library/windows/hardware/ff552325(v=vs.85).aspx).   
  **Note:** To create device metadata with the Device Metadata Authoring Wizard, select the **Internal device** option. After installing the tools, as described in [Getting started with Windows Store device apps](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265140(v=vs.85).aspx), the Device Metadata Authoring Wizard (DeviceMetadataWizard.exe) can be found at: %ProgramFiles(x86)%\Windows Kits\8.1\bin\
* The supplier develops and tests using that device metadata.
* The supplier hands off the sample or test app and device metadata to the OEM.

#### OEM integration

* The component supplier provides the appropriate components to the OEM – including their test device metadata, driver, and any sample source code for an app to access their device.
* The OEM creates their own device metadata that matches their PC and the app, based upon what the component supplier provides them.
* The OEM might need to repeat this process for each component supplier for which they plan to preload a Windows Store device app for a PC internal specialized device.
* If the OEM changes component suppliers, they need to change their app and mitigate any necessary deployment changes. A way to minimize deployment problems is to define a standard Device Interface Class contract for each component supplier.

#### Submission

Including the correct versions of all of the components in an OEM preload image is a good way to make sure all are present for bindings to succeed. For more information about OEM preload, see [Preinstall Apps Using DISM](http://go.microsoft.com/fwlink/p/?LinkId=325524).

The following steps show the deployment of an OEM-developed app. For the example of a custom IR blaster, the components involved include the device driver, device metadata, and the app developed by an OEM as an app preinstalled on a new tablet:

1. The component supplier passes driver and Hardware Certification tests, including Device.Devfund.CDA
2. The component supplier submits device driver to the Windows Dev Center hardware dashboard for certification and distribution via Windows Update.
3. The OEM passes the (Windows App Certification Kit) (WACK) tests for their app that incorporates functionality from the component supplier.
4. The OEM submits the app to Windows Store.
5. The OEM submits the device metadata package to the Windows Dev Center hardware dashboard for distribution via Windows Metadata and Internet Services
6. The OEM can now preload the certified and digitally signed device driver, device metadata, and app.

When submitting a privileged app for a PC internal device for Windows Store download, follow the instructions in [Publishing Apps for Windows Store Download](#_Publishing_Apps_for), but remember that the metadata in this case is the device metadata package for the PC, not for your device. If the app comes from a component supplier and not the OEM, the OEM needs to follow the OEM integration workflow, shown previously, before submitting device metadata.

## Publishing Device Drivers

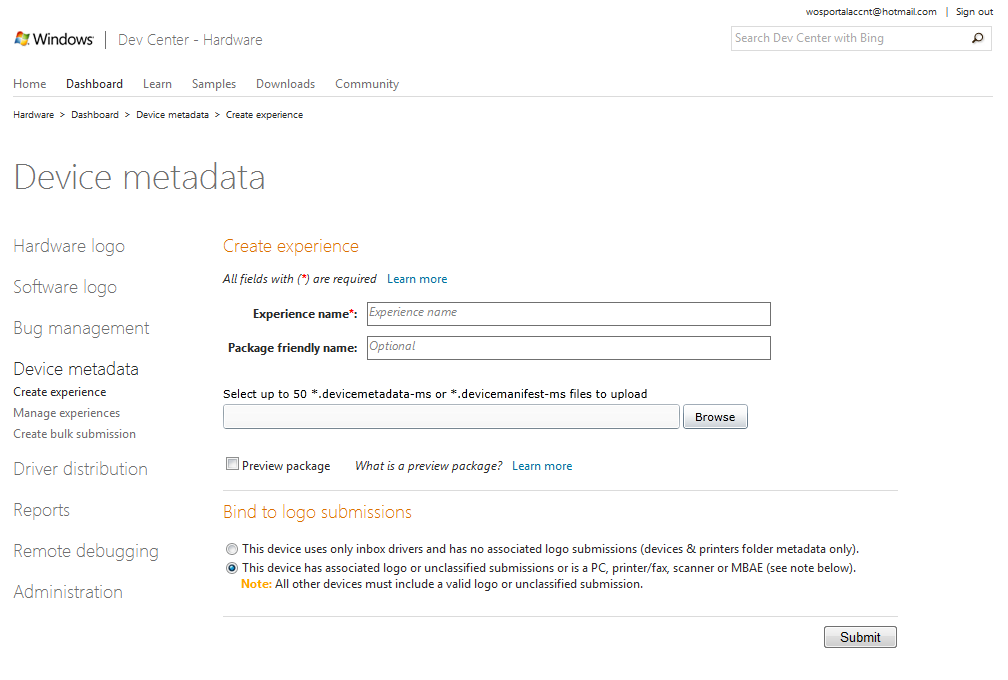
More information about authoring and validating drivers is available in [Developing, Testing and Deploying Drivers](http://msdn.microsoft.com/library/ff554651(v=vs.85).aspx) on MSDN®.

Your custom drivers must pass the appropriate Windows Hardware Certification Kit tests before submission. Once the drivers pass, they should be submitted to the [Windows Dev Center hardware dashboard](http://msdn.microsoft.com/en-us/windows/hardware/br257577) so they can be preinstalled. The user shouldn't be required to run any desktop installation program before your device driver is available.

Custom drivers should be submitted before your app or device metadata.

**IMPORTANT**   Device access using custom drivers requires approval from Microsoft. OEMs and IHVs that want to implement device access for a specialized device using a custom driver should first contact their Microsoft contact to discuss their scenario with the Windows Ecosystem team. For more information see the Custom Driver Access Model section.

Because your app needs custom access to a restricted device, when submitting device metadata, you need to choose "This device has associated logo or unclassified submissions …" under Bind to logo submissions.



This option requires that you provide the submission ID from the driver submission. This allows the dashboard to verify that the custom drivers have passed the Device.DevFundCDA portion of the Hardware Certification Requirements.

## Updating a Windows Store Device App for a PC internal specialized device

The process for updating a Windows Store device app is very similar to the process for initial deployment. And the unique elements described previously for specialized devices apply in both cases.

The components that come together to enable a Windows Store device app for a specialized device are loosely coupled – there's no tight version binding between them. Additionally the different components are distributed through different services, with different policies and latencies:

* App updates are available sometime after being accepted. Users are notified when app updates are available and can download them through the Windows Store app.
* Device metadata updates are available 48 hours after being accepted and are downloaded automatically for new devices connected to the PC. Device metadata for the PC is automatically updated by a Windows maintenance task every 14 days.
* Driver updates are available sometime after being accepted, depending upon Windows Update client settings. Users are notified of these optional updates if they check Windows Update, where they can download and apply the driver.
  + The driver can be available as soon as 24 hours, depending upon those settings.

Because of the availability time differences, it's likely that the app, device metadata, and custom driver may be of different versions. Components need to cope gracefully if any companion components are unavailable.

Only the OEM can deploy device metadata updates to the PC device container. All parties need to be in close communication about the contents of the privileged app fields and driver functional contracts. The component supplier can submit updates to the app and driver to Microsoft portals, but the OEM governs metadata. It's important to understand that Windows 8.1 doesn't automatically install a new driver over an old driver.

Aside from that, the sequence for updating your Windows Store device app is the same as the sequence you used to submit it.

# Design Patterns for Specialized Devices

This table summarizes design patterns for different types of devices. You can choose a design pattern based on the protocol used to communicate with the device and the type of device driver. The table also identifies which design patterns have restricted interfaces. The main steps in the planning and implementation of each design pattern and which Windows Runtime APIs to use are outlined after the table.

## Summary of PC Internal Specialized Device Design Patterns

|  |  |
| --- | --- |
| Design Patterns | |
| **Pattern: WPD with Custom Driver**  **Summary:**   * Intelligent device with functionality that fits into a well-defined object model with properties. * Device functionality can be represented as a collection of hierarchical storage and/or device services. * Applies to any app that interfaces with the WPD API set.   **Example:**   * A custom sensor. | |  |  | | --- | --- | | Protocols | Needs additional protocol layers beyond what's delivered in Windows 8.1. | | Transports | USB, TCP/IP | | Windows Runtime API | WPD APIs: [WPD Automation](http://msdn.microsoft.com/library/windows/desktop/dd389299%28v=VS.85%29.aspx),[Windows.Devices.Portable](http://msdn.microsoft.com/library/windows/apps/windows.devices.portable.aspx) | | Sample | [Portable device API sample](http://code.msdn.microsoft.com/Portable-Device-API-57c4e696) | | Restricted Interface | Yes | | Acquisition Options | Store Download | |
| |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Pattern: Custom Driver Access**  **Summary:**   * Use this pattern for a device that doesn't fit into the WPD Custom Driver pattern. * A custom device driver is required and a privileged applicationapp transfers information to and from the hardware via that device driver by sending IOCTL commands and receiving data.   **Example:**   * Custom IR blaster embedded in a tablet device. | |  |  | | --- | --- | | Protocols | Other | | Examples | Custom IR blaster embedded in a tablet device. | | Windows Runtime API | Device Access API | | Sample | USB: [Custom Driver Access sample](http://code.msdn.microsoft.com/windowsapps/Custom-device-access-sample-43bde679) | | Restricted Interface | Yes | | | |

**Note**: For patterns that require Bluetooth as a transport, we recommend using the Windows Runtime APIs for Bluetooth: [Windows.Devices.Bluetooth.Rfcomm](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.bluetooth.rfcomm.aspx) and [Windows.Devices.Bluetooth.GenericAttributeProfile](http://msdn.microsoft.com/en-us/library/windows/apps/windows.devices.bluetooth.genericattributeprofile.aspx). For more information, see [Supporting Bluetooth Devices](http://msdn.microsoft.com/en-us/library/windows/apps/dn264587.aspx).

## Pattern: WPD Custom Driver

|  |  |
| --- | --- |
| Windows Portable Device Platform | |
| This pattern applies to any app that interfaces with the WPD API set in the Windows Runtime.  Example: Custom Sensor | |  |  | | --- | --- | | Protocols | Custom Protocols | | Windows Runtime API | WPD APIs: [WPD Automation](http://msdn.microsoft.com/library/windows/desktop/dd389299%28v=VS.85%29.aspx),[Windows.Devices.Portable](http://msdn.microsoft.com/library/windows/apps/windows.devices.portable.aspx) | | Restricted Interface | Yes | | Acquisition Options | Store Download | |

The [Windows Portable Device (WPD) Platform](http://msdn.microsoft.com/en-us/library/windows/hardware/ff597729(v=vs.85).aspx) provides a flexible, robust way for apps to communicate with a variety of devices. Such a device may provide custom data or expose a service that enables an app to interact with custom functionality to retrieve data from the device.

The WPD Platform is designed to support many different kinds of specialized devices. It was initially developed to support smart storage devices such as portable media players and digital still cameras. With a WPD custom driver, it can support a multitude of other devices as well.

For devices that support proprietary protocols or custom services, you can build a WPD Custom driver to expose your device's capabilities to your Windows Store app using the WPD API SendCommand method. Refer to [IPortableDevice::SendCommand](http://msdn.microsoft.com/library/windows/desktop/dd375691(v=vs.85).aspx) method for more information.

Note that only restricted interfaces can access [WPD custom drivers](http://msdn.microsoft.com/library/windows/hardware/gg463541). Restriction in this case is per interface. This is particularly appropriate for devices that provide private user data, so information is exchanged only to privileged apps.

|  |  |
| --- | --- |
| WPD custom driver advantages | WPD custom driver disadvantages |
| * Can meet needs of a broad range of devices that implement protocols that are not natively supported in Windows. * Flexible API set. * Only the Windows Store device app has access to restricted interfaces. | * Developing a WPD custom driver can be costly if the developer isn't familiar with Windows driver development. * Requires driver certification and signing through the Windows Hardware Certification Program. |

### App

The app uses the WPD APIs to access device capabilities. The Windows Software Development Kit (SDK) includes examples of how the app uses the WPD APIs to interact with the device.

### Finding the Device

Before being able to access a device, the app must be able to find it. The device Object enumeration API is used to pass the device ID to WPD. A [device enumeration sample](http://code.msdn.microsoft.com/windowsapps/Device-Enumeration-Sample-a6e45169) demonstrates how this is done.

### Authoring

This checklist describes what to do when authoring the software components for your Windows Store device app that follows the WPD custom driver design pattern. Note the testing and validation requirements for each component.

|  |  |  |
| --- | --- | --- |
| Component | Item | Description |
| Device Driver | Implement custom functionality | The functionality for the custom device. |
|  | Follow security requirements | A restricted interface can't be opened directly from an app container. This is part of Hardware Certification Requirement Device.DevFundCDA. |
|  | Create a driver package | The driver package contains the driver binaries and the .inf file that lists the device IDs of the devices the driver should be installed on, and which driver binaries and/or registry settings are required for each. |
|  | Test and Validate | To submit the WPD custom driver to Windows Update, the driver must pass the Windows Certification tests for an unclassified device. |
| Device Metadata | Privileged Apps | This element provides the privileged app info for each app, including the Package Name and Publisher Name.  In the PC internal device case, the device metadata is bound to the PC itself using the PC's Hardware ID, and managed by the OEM. |
| App | Device Capability declaration | Define interfaces that the app will use in the app’s manifest file, Package.appxmanifest. |
|  | Device Enumeration | The app must select an instance of the device by using the Device Enumeration API. Use the Device.Enumeration API to query for instances of the Interface Class GUID, and then select an instance to open. |
|  | Device Access | C++ apps  IPortableDevice::Open: opens a handle to a device  IPortableDeviceService::Open: opens a handle to a device service  WWAs  PortableDeviceAutomation.Factory's GetDeviceFromIdAsync function: returns a device object that can be used to communicate with a device or device service. |
|  | Testing and Validation | Pass the app Certification Requirements test both with your device attached to the system and with your device disconnected from the system. |

### Submission

If the device requires a WPD custom driver, the driver needs to be submitted as an Unclassified Device driver. The device metadata also needs to be submitted for signing and distribution. Follow the instructions in [Building Windows Store device apps for Windows 8.1](http://go.microsoft.com/fwlink/?LinkID=246571), and see [Publishing Device Drivers](#_Publishing_Device_Drivers) in this document for more details on driver submission.

### Updating

Follow the instructions in [Building Windows Store device apps for Windows 8.1](http://go.microsoft.com/fwlink/?LinkID=246571) for additional details on updating a Windows Store device app. Here's how you to submit updated components so that Microsoft distribution services can make them available to end users.

#### Device driver updates

* As you revise your driver, be sure that it's compatible with the published version of your app. If you add or change an interface, the app also will need to be modified.
* Submit the driver for Certification and signing on the Windows Embedded Standard (WES) portal and grant Windows Update redistribution rights.

#### Privileged app updates

* App updates require coordination between the app publisher and the device metadata author to ensure that the metadata contains the proper references.
* Updated apps need to maintain the same fields that are listed in the privileged app list (Package Name and Publisher Name).

#### Device metadata updates

* Metadata updates require coordination between app publisher and the owner of the PC container to ensure that the metadata contains the proper references.
* Metadata must be submitted and signed before it can be trusted and distributed by Microsoft.

## Pattern: Custom WDF Driver

|  |  |
| --- | --- |
| Custom Driver Access | |
| A device that doesn't fit into any of the patterns previously mentioned in this document.  A custom device driver is required and a privileged app transfers information to and from the hardware via that device driver by sending IOCTL commands and receiving data.  Example: Custom IR blaster embedded in a tablet device. | |  |  | | --- | --- | | Protocols | Other | | Windows Runtime API | Device Access API | | Sample | USB: [Custom Driver Access sample](http://code.msdn.microsoft.com/windowsapps/Custom-device-access-sample-43bde679) | | Restricted Interface | Yes | |

With this pattern, you can talk to any type of device by letting your app talk to the custom device driver for your device. Use this pattern when none of the other patterns is a good match for your device. It was designed to enable new, innovative devices for which built-in support may not exist for years. It provides the most flexibility to craft your solution, but also requires the most development effort and the most testing, and may have higher support costs over time due to the number of different components required. There are additional limitations listed in the following table.

**IMPORTANT**

* Device access using custom drivers requires approval from Microsoft. OEMs and IHVs that want to implement device access for a specialized device using a custom driver should first contact their Microsoft contact to discuss their scenario with the Windows Ecosystem team. For more information see the Custom Driver Access Model section.
* Windows 8.1 introduces new Windows Runtime device protocol APIs that Windows Store apps can use to access USB, Human Interface Devices (HID), Bluetooth GATT, Bluetooth RFCOMM, Wi-Fi Direct devices, and more. For more info, see [Windows 8.1: New APIs and features](http://go.microsoft.com/fwlink/?LinkID=298951).
* OEMs can access PC internal devices connected to the PCs USB bus by providing a Windows Store app privileged access to the Device Container representing the PC. Accessing PC Internal devices using one of these methods is preferred to creating a custom driver and does not require approval from Microsoft. Only privileged apps designed by the OEM for the Device Container may access PC internal devices using the USB and HID APIs. For more information, see [Building Windows Store device apps for Windows 8.1](http://msdn.microsoft.com/en-us/library/windows/hardware/dn265148(v=vs.85).aspx).

This pattern requires all four components: device, app, device metadata, and custom device driver with a Windows Runtime device component.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| * Can meet needs of the broadest range of devices. * For devices that already have Windows support, the corresponding custom device drivers are most easily adapted to this pattern. * Vendor controls which Windows Store apps access their device. | * Costly. Requires development and maintenance of a third-party driver. * Complex. This is the most complicated pattern with the most components and moving parts. * App restrictions. * Only designed for custom drivers installed against a specific hardware device. |

See [Building Windows Store device apps for Windows 8.1](http://go.microsoft.com/fwlink/?LinkID=246571) and also [Appendix A: Design Guidance for Custom Driver Access](#_Appendix_A:_Custom), to get a good idea of what's involved in developing the components for a Windows Store device app that follows the Custom Driver Access design pattern.

**Note:** This section describes the planning and creation process for a new device. If you have an existing device and/or device driver, you need to alter them to conform to the requirements listed here if they don't already meet them.

### Samples

SDK sample: [Custom Driver Access](http://go.microsoft.com/fwlink/p/?LinkId=258235) (previously known as the MoFX2 sample).

WDK sample: WUDF OSR USB FX2 sample. This sample contains four different versions of the driver, the simplest of which is in the WUDFOsrUsbFx2 project (but any of them should work for this sample). The sample also contains device metadata to enable the app located under the devicemetadatapackage in the solution directory.

### Planning and Design

#### Device

The custom device access pattern puts no restrictions on the device itself. For a Windows Store device app to work, the device must be discoverable by Windows and Windows must have the correct drivers to enable communication with it. Beyond that, there may be device-type specific requirements – see the Windows documentation for your device type for those.

For this pattern, the device's device ID generally derives from the underlying bus. Take, for example, the OSR USB Fx2 device (used in the sample app and driver listed previously). USB devices can have multiple hardware IDs, which are derived from the vendor ID (VID), product ID (PID), and Revision fields in the USB device descriptor. The hardware IDs for the OSR USB Fx2 device are:

* USB\VID\_0547&PID\_1002&REV\_0000
* USB\VID\_0547&PID\_1002

For a multifunction device, the container ID for the device could derive from the device firmware, or from the system firmware in the case of an internal device. A USB device can provide the Microsoft ContainerId operating system descriptor, and Windows automatically assigns that container ID to all of the subfunctions of the device.

See the documentation for your type of device and the bus it connects to for the exact way device and container IDs are formed.

#### Driver

This pattern requires a custom device driver for your device. As described above, the driver controls the device and provides your app with a way to communicate with the device. Your driver is responsible for publishing a device interface with a custom GUID so that your app can find the driver and communicate with it. As an example, the custom interface published by the OSR Fx2 device driver is {573E8C73-0CB4-4471-A1BF-FAB26C31D384}.

Your driver must mark the device interface as restricted, because the custom device access pattern doesn't support open access. If the interface isn't marked restricted, your app can find your device interface, but it can't open it. A driver marks the device interface by setting the DEVPKEY\_DeviceInterface\_Restricted property on the interface to TRUE. The sample Fx2 driver demonstrates the recommended way of doing this in the driver in code, but the property can also be set by the INF.

The device interface your driver creates must be accessible by a standard user running your app. For most drivers, using the default security settings provided by the system suffices. For drivers that apply custom security, ensure that you provide the appropriate level of access. This security descriptor string is an example:

D:P(A;;GA;;;SY)(AA;;GA;;;BA)(A;;GRGWGX;;;;WD)

This allows normal users to read, write, and traverse namespaces on your device while giving the system and administrators full access.

Unlike desktop apps, you can't simply grant the Administrators group access to the device and rely on users being signed in to an account with those access rights. The DeviceAccess APIs work under the access rights of the Users group. See [SDDL for Device Objects](http://msdn.microsoft.com/library/windows/hardware/ff563667(v=vs.85).aspx) for more information.

If you have additional device functionality that you want to expose to a desktop app, you can create multiple device interfaces with different GUIDs. Only the restricted interfaces are accessible by your Windows Store device app. Your desktop app can potentially access all interfaces, depending on security settings. Your driver should set different reference strings on each device interface. You should then use them to keep track of whether the client opened the restricted or non-restricted interface, so that your driver can keep the Windows Store device app from issuing I/O requests that are only allowed from the desktop context.

#### Device Metadata

Your device metadata binds to your device using the Hardware ID for the PC.

Because this pattern uses a restricted interface, your device metadata must list the apps that have access in the list of custom privileged apps. If it doesn't, your app can't open the device interface.

#### App

In the custom device access pattern, the app finds the device by searching for instances of its custom device interface. To do this, create a device interface search query that uses the custom GUID for your device and looks for currently present instances of the interface. A search query for the OSR USB Fx2 device's interface looks like this:

System.Devices.InterfaceClassGuid:="{573E8C73-0CB4-4471-A1BF-FAB26C31D384}" AND System.Devices.InterfaceEnabled:=System.StructuredQueryType.Boolean#True;

To communicate with the device, your app opens one of the device's interfaces using the Windows CreateDeviceAccessInstance API. This returns an object that allows your app to open the device interface and then use custom I/O control requests (IOCTLs) to have the driver perform actions on the device. If you're writing your app in C++, you can do this directly from within the app. You can also use the Windows.Devices.Custom.CustomDevice.FromIdAsync method to open your device and the SendIOControlAsync method to transfer information to and from the device.

An app can't simply access any device interface it chooses. The app must declare the device interfaces that it accesses in the capabilities section of the app manifest. For the Windows Store device app associated with the Fx2 device, this looks like:

<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">

…

<Capabilities>

<DeviceCapability Name="573E8C73-0CB4-4471-A1BF-FAB26C31D384" />

</Capabilities>

</Package>

### Development Lifecycle

#### Authoring

This checklist describes what to do when authoring the software components for a Windows Store device app that follows the custom driver access design pattern. Note the testing and validation requirements for each component.

|  |  |  |
| --- | --- | --- |
| Component | Item | Description |
| Device Driver | Provide a functional contract through a device interface. | The driver provides a custom device interface class GUID for its functional class. |
| Restrict the device interface. | The DEVPKEY\_DeviceInterface\_Restricted property should be set to TRUE on the interface instance of that class. |
| Implement custom functionality. | The driver provides IOCTL commands to transfer information to and from Windows Store apps. |
| Follow security requirements. | The driver shouldn't specify a security descriptor that grants the AppContainer access to the device. Instead, the driver should grant normal unelevated users access to the device, and rely on the Device Broker to mediate the App's access. |
| Create a driver package. | The driver package contains the driver binaries and the .inf file that lists the hardware IDs of the devices the driver should be installed on, and which driver binaries and/or registry settings are required for each. |
| Test and validate. | To submit a custom driver to Windows Update the driver must meet the Certification Requirements for your type of device.  For a custom access device, the driver must also pass Device.DevFundCDA. Note that this test requires manual intervention, which may increase the costs of testing. |
| Device Metadata | PrivilegedApps | This element provides the privileged app info for each app, including the Package Name, Publisher, and Access Custom Driver fields. Each privileged app identified checks the Access Custom Driver checkbox so it can communicate with the device using the custom device driver.  In the PC internal device case, the device metadata is bound to the PC itself, using the PC's Hardware ID, and managed by the OEM. |
| App | DeviceCapability declaration | Custom Device Interface Class GUID |
| Device Enumeration | The app must select an instance by using the Device Enumeration API. Use the Device.Enumeration API to query for instances of the Custom Device Interface Class GUID, and then select an instance to open. |
| Device Access | When a device instance is selected, communicate with the device by using CreateDeviceAccessInstance() to open a handle to the device and IDeviceIoControl() to transfer information to and from the device. You can also use the Windows.Devices.Custom.CustomDevice.FromIdAsync method to open your device and the SendIOControlAsync method to transfer information to and from the device. |
| Testing and Validation | Pass the app Certification Requirements tests both with your device attached to the system and with your device disconnected from the system. |

#### Submission

For this pattern you must submit drivers, apps, and device metadata. Pay special attention to the workflow described for components suppliers and OEMs listed previously in this document.

#### Updating

When updating the custom device driver, you must take care to maintain compatibility between versions of the driver and the app. See [Appendix A](#_Appendix_A:_Custom) for more details on how to do this for a custom device driver.

#### Example

In this example, a custom IR blaster is embedded into a tablet that includes a custom driver and a privileged app that has custom driver access.

The OEM preinstalls the device driver, device metadata, and app using the ADK.

When a user first boots a new tablet that was preinstalled with a device and app, setup searches Windows 8.1 for a matching device driver by hardware ID, and then install the device driver. Next, Windows 8.1 prompts the user to opt in to default settings and completes the out-of-box experience. The user signs in to the PC for the first time, and the preloaded device metadata is installed using a series of generated hardware IDs for the PC container derived from information in the system BIOS. The app is installed for that user, who then sees the app tile on their Start screen, and clicks it to open the app.

The metadata package bound to the PC lists the app in the list of privileged apps, and then sets the AccessCustomDriver attribute to TRUE. This allows the app to access any restricted interfaces within the PC container that the app has declared in its app manifest.

The driver creates a single device interface for the device, and marks the interface restricted so that the app can access it. This interface supports a few basic operations; foremost it allows the app to wait for an IR code to be received so that the app can then retrieve the associated data. Because the driver only creates a single interface for the device, it doesn't need to set a reference string on the interface.

The app is a JavaScript/HTML app that contains a C++ device projection library to access the device. The app queries for the device's custom interface, and when it finds an enabled interface it retrieves its path to create an instance of the device projection. The device projection object supports a single method to read the IR codes, which the JavaScript app can invoke. You can also use the Windows.Devices.Custom.CustomDevice.FromIdAsync method to open your device and the SendIOControlAsync method to transfer information to and from the device.

# Appendix A: Custom Driver Access Design Guidance

The custom driver access design patterns are the most complex and costly design patterns presented in this document. Here's design guidance for planning each software component so that you can provide the best experience to customers.

## Authoring a Device driver

You should write your custom drivers using the User Mode Driver Framework (UMDF) wherever possible. UMDF reduces the impact to users if a driver crashes or if it's compromised by an attacker. Custom WPD drivers already make use of UMDF.

If you must write a kernel-mode driver, use the Kernel Mode Driver Framework (KMDF); don't write a traditional Windows Driver Model (WDM) driver. KMDF dramatically simplifies the process of writing a kernel-mode driver, and KMDF drivers tend to be more reliable than WDM drivers because of this.

If the device fits into a defined class, be aware of the Hardware Certification Requirements and of the Device Fundamentals requirements. The Windows Driver Kit (WDK) enables you to run a number of Device Fundamentals tests against your driver. The WDK also contains code analysis tools to scan drivers for defects before you start testing.

## Communicating Between App and Driver

If you're creating a custom device driver, you'll be responsible for defining the interface between the app and the driver. The driver writer and app developer need to be in close communication about the functional contract of the IOCTL interfaces.

Create and register a Custom Device Interface Class GUID that represents the functional contract of your hardware's unique functionality. Set the restricted property on any instances of the interface class to make them accessible to your app.

If you introduce a Custom Device Interface Class GUID representing a functional contract, you must maintain that functionality forever, because apps depend upon that behavior. If you need to add additional functionality in subsequent revisions of your driver, introduce a new Custom Device Interface Class GUID that represents that additional functional contract.

Don't use custom PnP events to communicate with your Windows Store app – it can't receive these messages. Instead, use pending asynchronous I/O, if you need to send asynchronous events, like notification of button presses, to your app. The Custom Device Access sample shows how to do this.

We recommend of that you test your IOCTL interfaces initially using a desktop (Windows API) app, because it's easier than using a Windows Store app.

## Coordinating App and Driver Updates

Updating a driver that's used in a Windows Store device app can pose some challenges. In particular, if the update contains new support that the app depends on, you must take special precautions to maintain compatibility.

Apps and drivers are distributed from different online services and are picked up by users at different rates. Additionally, Windows 8.1 doesn't automatically install a new driver over an old driver – the user must use Windows Update and select the optional driver update. As a result, it's possible for an app and its driver to be updated separately and to be different versions.

The app and driver must be compatible with each other both backward and forwards to provide a good user experience in all cases. Make sure that any updates are written to be completely compatible with all previous versions of the driver that your app has ever supported. For example:

* Version 1 of a driver introduces a functional contract as Custom Device Interface Class represented by "A".
* Version 1 of the corresponding app is written to the driver functionality represented by "A".
* Version 2 of the driver maintains the functionality represented by "A" and adds new functionality with a new functional contract as Custom Device Interface Class represented by "B".
* Version 2 of the app takes advantage of the new functional contract that "B" represents.
* Version 2 of the app works with functional contract "A" if a driver with functional contract "B" isn't present.

## Submitting Updates

When submitting an update that only changes one component, and which doesn't change the privileged apps, you can simply submit that component to the relevant service. When updating multiple components, submit them in the same order described previously in [Windows Store device app development lifecycle](#_Metro_Style_Device).

# Appendix B: Porting existing desktop device apps to Windows Store device apps

Windows Store apps are fundamentally different than desktop apps. Many desktop device apps depend on APIs or capabilities that aren't available in the new environment for Windows Store apps. Your existing desktop app simply may not function in the new immersive environment. See [Getting started with Windows Store apps](http://msdn.microsoft.com/en-US/windows/apps/br211386) and [Concepts and architecture](http://msdn.microsoft.com/en-us/library/windows/apps/br211361.aspx) for more information.

Windows 8.1 controls how and when Windows Store apps run. Windows runs the Windows Store app in the foreground and pauses the apps in the background. This allows Windows Store apps to be fast and fluid and helps Windows conserve power. See the [Fundamentals of Windows Store apps: How and when your app will run](http://channel9.msdn.com/Events/BUILD/BUILD2011/APP-409T) presentation and [App Lifecycle](http://msdn.microsoft.com/en-us/library/windows/apps/hh464925.aspx) for more information. When it's in the background, there's no way a Windows Store device app can wake up and respond to a PC internal specialized device–related event.

Windows Store apps have restricted access to resources such as the network, registry, and local storage. The app declares its required capabilities in its manifest, and customers can view them in the Windows Store before purchase and in the Settings flyout after installation. A Windows Store app can't access resources that aren't owned by the user, and even then only if they're granted access.

Windows Store apps can't access or modify system settings through the registry or Windows Management Instrumentation (WMI). Desktop apps that depend on changing shared system settings in either of those ways aren't appropriate for Windows Store app development. Services aren't part of the Windows Store app platform or available to Windows Store device apps. Unless a Windows API can provide access, the app can't communicate with services running on the same PC, either directly through Remote Procedure Call (RPC) or Distributed Component Object Model (DCOM), or through the network loopback. This reinforces the per-user aspect of a Windows Store app.

Control Panel and desktop management apps are alternatives to a Windows Store device app if system management capabilities are needed.

# Appendix C: Number of Apps Policy

Device manufacturers are limited in the number of Windows Store apps that may be specified in device metadata for automatic installation and app privilege. For example, peripheral device manufacturers (IHVs) can submit up to one app that is configured for automatic installation and up to one app that is specified as a privileged app. An IHV can submit one app that meets both limitations or two apps, with each meeting just one of the limitations.

Mobile operators and OEMs have different limits on the number of apps that they can specify in device metadata. For more info, OEMs should contact their Microsoft OEM representative.

In each device metadata package, the following limits apply:

|  |  |  |
| --- | --- | --- |
| Developer | Automatic installation app limit | Privileged app limit |
| IHV (externally connected peripheral device) | 1 | 1 |
| Mobile operator | 1 | 8 |
| OEM | contact Microsoft | contact Microsoft |

**Important** There is no limit to the total number of Windows Store device apps that a device manufacturer can submit to the Windows Store; these limits apply only to a single device metadata package.

# Glossary

Here are some definitions for terms used in this document.

|  |  |
| --- | --- |
| Term | Definition |
| custom driver | A KMDF or UMDF driver supplied by the hardware developer (as opposed to an in-box class driver supplied by Windows). |
| Device ID | An identifier that describes either the make and model of a device (hardware ID) or the industry standard device class (compatible ID). |
| live tile | A tile is a representation of an app on the Start screen. Clicking or tapping a tile on a touch-enabled device launches the app. Tiles that update with new information are live tiles. The content shown on tiles can, and ideally should, change regularly, especially if it can communicate new info in real time to users. Tiles can show text, images, and a badge to show status. |
| Windows Store app | An app built using the new app model in Windows 8.1. These apps are built using Windows Runtime APIs. |
| Windows Store device app | A Windows Store app associated with a device by using device metadata. This app can be automatically acquired from the Windows Store (when associated with an externally-connected peripheral device). |
| splash screen | A composite of a splash screen image and a background color, both of which can be customized. It's displayed immediately when an app is launched and is dismissed when the first view of the app is ready for interaction, providing valuable feedback to users. |
| toast | A toast notification is a transient message to users that contains relevant, time-sensitive information and provides quick access to related content in an app. It can appear while another app is in use, and on the Start screen, the lock screen, or the desktop. |