Componentization of Device Drivers in Windows Embedded Standard 7

Introduction

This article describes the componentization of in-box device drivers for **Windows Embedded Standard 7 (Standard 7)**. Within this article, any mention of “driver” refers to an “in-box” device driver which is installable from Standard 7.

Device driver support is very important for embedded devices, especially because many embedded devices are designed to be used with specific or specialized hardware. However, there are cases in which off-the-shelf hardware is used based on availability or to reduce the cost per device. Regardless, robust and reliable hardware functionality is very important for the successful deployment of embedded devices. This leads to the question, “Why Componentize Drivers”?

Why Componentize Drivers?

Because many embedded devices are constrained by storage space, it would be wasteful to have all the driver files present on the device. Instead, it would make sense for the device to only contain the driver files that are required for the specific hardware on the device. In Windows 7, all the driver files are present on the destination computer regardless of which drivers are actually loaded.

In addition to the reduction in storage space, we receive other benefits which include the following:

- **Reduced maintenance overhead** – updates and service packs only contain the necessary driver changes.
- **Reduced attack surface** – from a security standpoint, the device can be locked down by intentionally omitting driver software for hardware that has to be disabled on the device (for example, disabling USB support).
- **Streamlined installation images** – after you create an embedded image, only the required or desired driver software will be present.

Some driver installation scenarios are enumerated in the following section.

Driver Installation Scenarios

- Quickly create and deploy an image on an embedded device by using the **Imaged Based Wizard (IBW)**, where we are prototyping. We want to automatically discover all the hardware attached to the device and install the correct drivers for them.
- Create an “answer file” image configuration by using the **Image Configuration Editor (ICE)**, where we only want to include drivers for the hardware we want to support.
- Quickly create and deploy an image to an embedded device by using IBW, but instead of automatically discovering the hardware attached to the device, we want to import a

DEVICES.PMQ file. This can be generated by running **Target Analyzer Probe (TAP.EXE)**. We have edited the PMQ file to remove the hardware entries we do not want to support.

- After you deploy an image to a device, you want to install additional drivers online after booting into it.

These scenarios were central to the design of this solution.

The first step is determining the exact set of drivers we have to componentize.

Identifying Drivers to Componentize

Standard 7 is based on the **Windows 7 Ultimate** product. So, the set of drivers to componentize include all those present in Ultimate. However, certain drivers must always be present in an embedded image. Therefore they must always be installed automatically. These drivers fall into the following two categories:

- **Boot critical drivers** – these drivers are considered required for an embedded image to boot. In Windows, each driver is categorized in a "device class". For boot critical drivers, the associated device classes are considered boot critical.
- **Bus enumerator drivers** – these boot critical drivers manage and discover hardware attached to a device bus (for example, PCI, USB, AGP, and more.). Because these drivers are necessary for device discovery, they must always be present.

Therefore, the boot critical and bus enumerator drivers will be installed as part of the Standard 7 runtime, with the remaining drivers being the exact set we have to componentize. Because we now have the set of drivers to process, we now have to examine the driver data we must collect. A driver is installed from an INF file. Upon close investigation of INF files, we find that we can extract all the driver data we need from them.

The following section examines the INF file structure to see where the data resides.

INF File Structure

INF files are basically organized into "sections", where each section contains one or more key/value pair "entries". Here is an example INF section with some entries. For detailed information on the INF file structure see About INF File Architecture.

```
[Version]
Signature="$Windows NT$"
Class=MEDIA
ClassGuid={4d36e96c-e325-11ce-bfc1-08002be10318}
Provider=Microsoft
DriverVer=07/13/2009,6.1.7600.16385
PnPLockdown=1
```

INF files contain lots of data for driver installation. However, we will only focus on what we need. In particular, the following information is of interest:

- **Device identification strings**

- **Include entries**
- **Package-aware printer driver sharing**

Device Identification Strings

The Plug and Play (PnP) manager and Windows setup use “device identification strings” to identify hardware which is attached to a computer. There are three types of device identification strings. These are as follows:

1. **Device ID** – a given piece of hardware has only one device ID that is the most specific “hardware ID” for it.
2. **Hardware ID** – is a vendor-defined identification string that Windows setup uses to match a hardware device to an INF file. There is usually a list of hardware IDs for a given hardware device, where the first is the device ID and the subsequent IDs are less specific.
3. **Compatible ID** – is also a vendor-defined identification string that Windows setup uses to match a hardware device to an INF file. These are used by Windows setup to match a hardware device in case all the hardware IDs did not provide a match.

So Windows setup maps discovered device identification strings to driver INF files when it tries to find a driver. The best match is found by matching the device ID first. When this is not matched, the remaining hardware IDs are matched followed by the compatible IDs. For a complete discussion on how drivers are ranked and matched see How Setup Selects Drivers.

Device identification strings are defined in the INF “models” section, here is a sample of a models section where the highlighted areas are device identification strings; in this case they are all hardware IDs. If you want to dig deeper into INF sections see INF File Sections and Directives.

```
[ATI.Mfg.NTx86...1]
"AMD 760G (Microsoft Corporation WDDM 1.1) " = ati2mtag_RS780, PCI\VEN_1002&DEV_9616
"AMD 780E (Microsoft Corporation WDDM 1.1) " = ati2mtag_RS780, PCI\VEN_1002&DEV_9615
"ATI FireGL T2 (Microsoft Corporation - WDDM) " = ati2mtag_RV350GL, PCI\VEN_1002&DEV_4154
"ATI FireGL T2 Secondary (Microsoft Corporation - WDDM) " = ati2mtag_RV350GL, PCI\VEN_1002&DEV_4174
"ATI FireGL V3100(Microsoft Corporation - WDDM) " = ati2mtag_RV370GL, PCI\VEN_1002&DEV_5B64
"ATI FireGL V3100 Secondary (Microsoft Corporation - WDDM) " = ati2mtag_RV370GL, PCI\VEN_1002&DEV_5B74
"ATI FireGL V3200 (Microsoft Corporation - WDDM) " = ati2mtag_RV380GL, PCI\VEN_1002&DEV_3E54
"ATI FireGL V3200 Secondary (Microsoft Corporation - WDDM) " = ati2mtag_RV380GL, PCI\VEN_1002&DEV_3E74
"ATI FireGL V3300 (Microsoft Corporation - WDDM) " = ati2mtag_RV515GL, PCI\VEN_1002&DEV_7152
```

Include Entries

INF files can “include” other INF files to refer to common sections and entries. This is defined in the “DDInstall” section with the “Include” entry.

What follows is an example DDInstall section where two INF files are being included. The highlighted line shows the Include entry; in this case both the “ks.inf” and “wfmaudio.inf” files are being included.

```
[HdAudModel]
Include=ks.inf,wdmaudio.inf
```

```
Needs=KS.Registration,WDMAUDIO.Registration,mssysfx.CopyFilesAndRegister
CopyFiles = HdAudModel.CopyList
AddReg    = HdAudModel.AddReg
AddProperty = HdAudBranding.AddProperty, HdAudModel.AddProperty
```

Package Aware Printer Driver Sharing

There is another mechanism used to share INF sections and entries. However, it is specific to printer driver INF files. To optimize the writing of printer drivers, Microsoft provides a "core printer driver" INF named "ntprint.inf". This specifies common sections and entries. Printer driver INF files can be coded to refer to this common data by using the "Package Aware Printer Driver Sharing" feature.

The common sections and entries are shared with GUIDs from ntprint.inf. To reference these common sections and entries, a printer INF defines the "CoreDriverSections" entry in the DDInstall section, where the common sections and GUIDs are referenced. Shown here is an example in which the highlighted line shows this, the sections: "UNIDRV.OEM", "UNIDRV_DATA" and "sRGBPROFILE.OEM" are located in ntprint.inf. All the entries within these sections are also implicitly shared. Printer driver writers can use this feature to share sections and entries within their own printer drivers. If you are interested in reading more see Package Aware Printer Drivers That Share Files.

```
[BRM8640D.LPT.GPD]
CopyFiles=@BRM8640D.GPD,BRZL2_UNIDRV.CopyList
DataFile=BRM8640D.GPD
CoreDriverSections="{D20EA372-DD35-4950-9ED8-A6335AFE79F0},UNIDRV.OEM,UNIDRV_DATA","{D20EA372-DD35-4950-9ED8-A6335AFE79F3},sRGBPROFILE.OEM"
```

Driver Packaging and Selection

Now that we have identified all the required driver data, we can now focus on how we will package the drivers. We also have to figure out how to select the correct drivers for installation.

We will create driver packages which will contain all the driver bits which must be installed. Also, we will provide package metadata in the driver packages which will be used to identify them for specific hardware devices. We also have to think about how we will manage dependencies, because a driver might require other driver or non-driver software to be previously installed. We can add this dependency information into the package metadata also.

We take these ideas and produce a design for driver packaging.

The following section examines the proposed driver package structure.

Driver Package Structure

Each driver to be componentized will be in its own driver package. This driver package structure will consist of the following:

- **Driver payload**
- **Package manifest file**
- **Package catalog file**
- **Driver component manifest file**

Driver Payload

This is composed of the driver INF file and all the driver specific files such as SYS, DLL, and so on. – note some driver payload will only consist of an INF file. There will also be an additional catalog file for printer drivers only. This file contains a hash of the complete printer driver payload.

Package Manifest File

We will be using the standard Windows package format for our packages. Therefore, we have to provide a package manifest file. This file specifies internal package settings required by Windows setup. This file can be augmented to include additional settings also. We will take advantage of this by adding the package “metadata” we mentioned earlier. This package metadata will consist of the following:

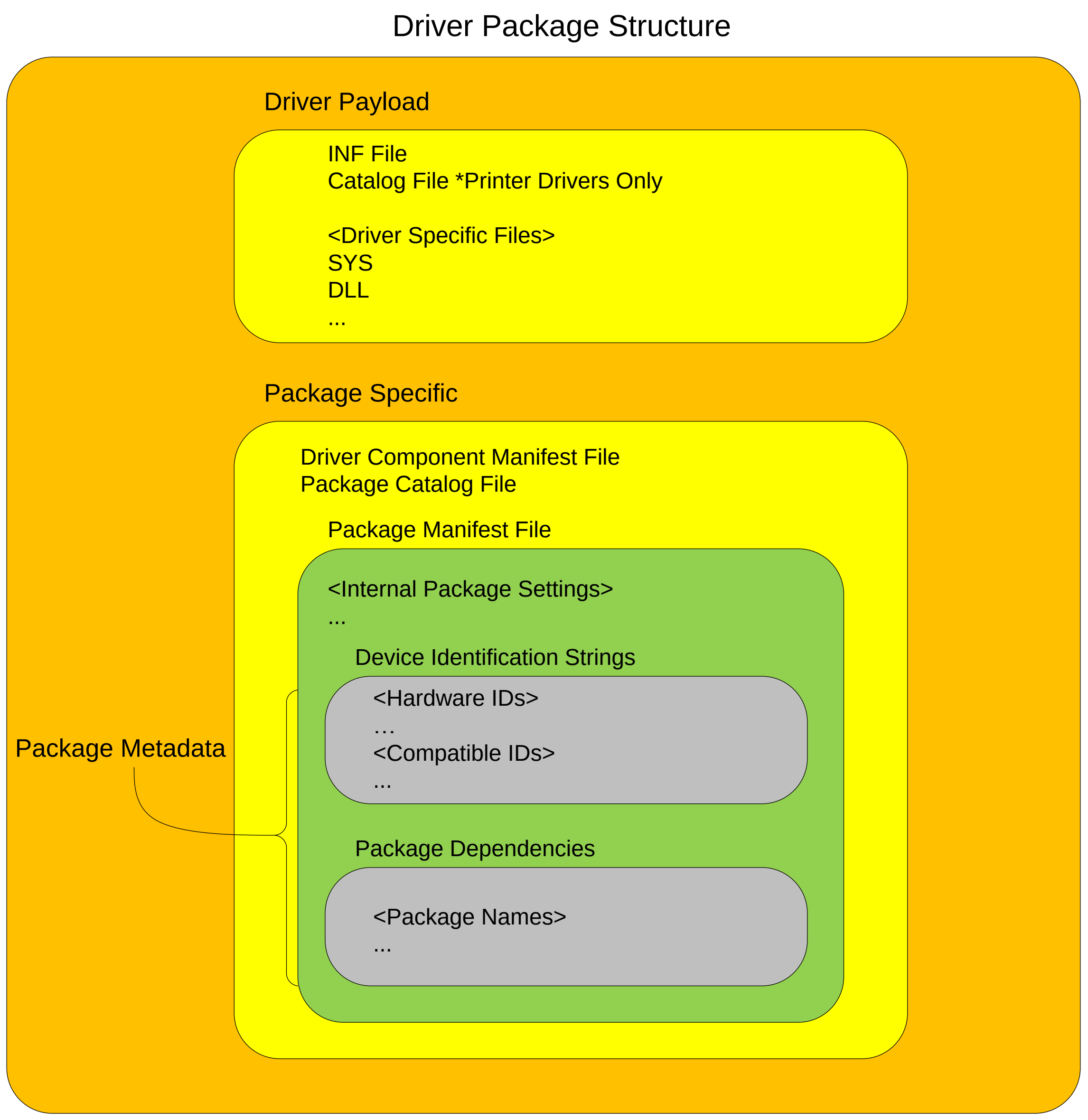
1. **Device identification strings** – this will be all the hardware and compatible IDs for all the hardware devices supported by the driver INF. This will be used by IBW and ICE to map discovered device identification strings for attached hardware devices on an embedded device, to driver packages.
2. **Package dependencies** – this is the list of driver or non-driver packages. These must be installed before this driver Package. This information will be used by IBW and ICE.

Package Catalog file

This is a required file for all Windows packages. It contains a hash of the complete driver package.

Driver Component Manifest File

Within Windows, each driver is considered a “component”. Components are referenced from packages, and are the smallest unit of installation within Windows setup. Each component is defined in a “component manifest file”.



Now that we have the package format that is defined and that supports the desired driver installation scenarios we have previously outlined, we can now focus on how to build these packages.

Driver Package Creation Process

Here are the steps we have to take:

1. **Parse the INF files**
2. **Determine the package dependencies**
3. **Store the processed driver data**
4. **Build the packages**
5. **Publish the packages**

Parsing the INF Files

As mentioned earlier, the Standard 7 product is based on Ultimate. Therefore all the relevant INF files from Ultimate are parsed. However as we mentioned earlier, we will filter out any INF files for drivers which are boot critical or bus enumerators. These drivers will always be installed with the Standard 7 runtime.

For each INF file we process, we extract the following data:

- **INF filename**
- **Device class**
- **Include entries**
- **Package Aware Printer Driver Sharing entries**
- **Hardware IDs**
- **Compatible IDs**

The extracted hardware and compatible IDs will be added to the package metadata.

Determining the Package Dependencies

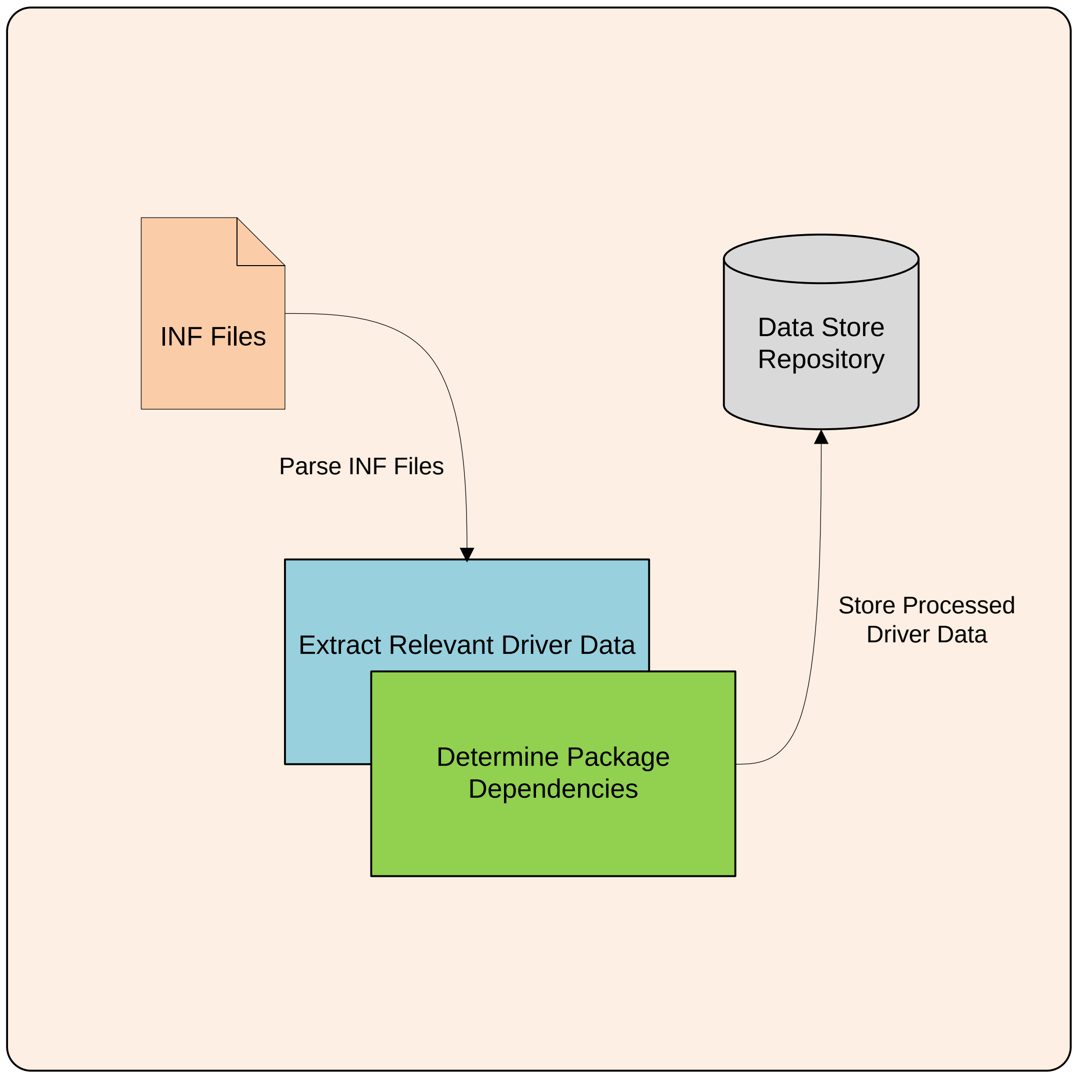
After we have parsed the required INF files to pull the relevant data, we must now determine whether we have any package dependencies. Therefore, we can add them to the package metadata. There are three types of dependencies:

1. **Driver dependencies** – these can be discovered by examining the Include entries and by looking at the Package Aware Printer Driver Sharing entries. The result will be referenced INF files, where each referenced INF refers to a driver package we must add a dependency on.
2. **Binary dependencies** - for driver binary payload files (for example SYS, DLL files and so on) we perform binary dependency analysis. We examine each driver binary to see whether it depends on other files in the system. The packages which contain these discovered file dependencies are added as package dependencies. These can be driver or non-driver package dependencies.
3. **Undiscoverable dependencies** – there are driver and non-driver package dependencies which are not discoverable. In this case, we manually add these package dependencies to the package metadata.

Storing the Processed Driver Data

All the processed INF data is stored in an internal “data store” repository that contains all the extracted driver information.

Driver Package Creation Process – Steps 1 thru 3


INF Files
Data Store Repository
Parse INF Files
Store Processed Driver Data
Extract Relevant Driver Data
Determine Package Dependencies

Building the Packages

There is an internal **package build** tool that is used to build packages. This tool was also adapted to build driver packages. The source for packages is defined in "YAML" files, which define a simple key/value syntax. See YAML Ain't Markup Language for more information on the YAML file format.

The YAML files for driver packages are automatically generated for each driver defined in the data store repository. What follows is an example YAML file for the **"**WinEmb-INF-hdaudio**"** driver package. Note: the "$(MWE)" macro expands to the "WinEmb-" prefix.

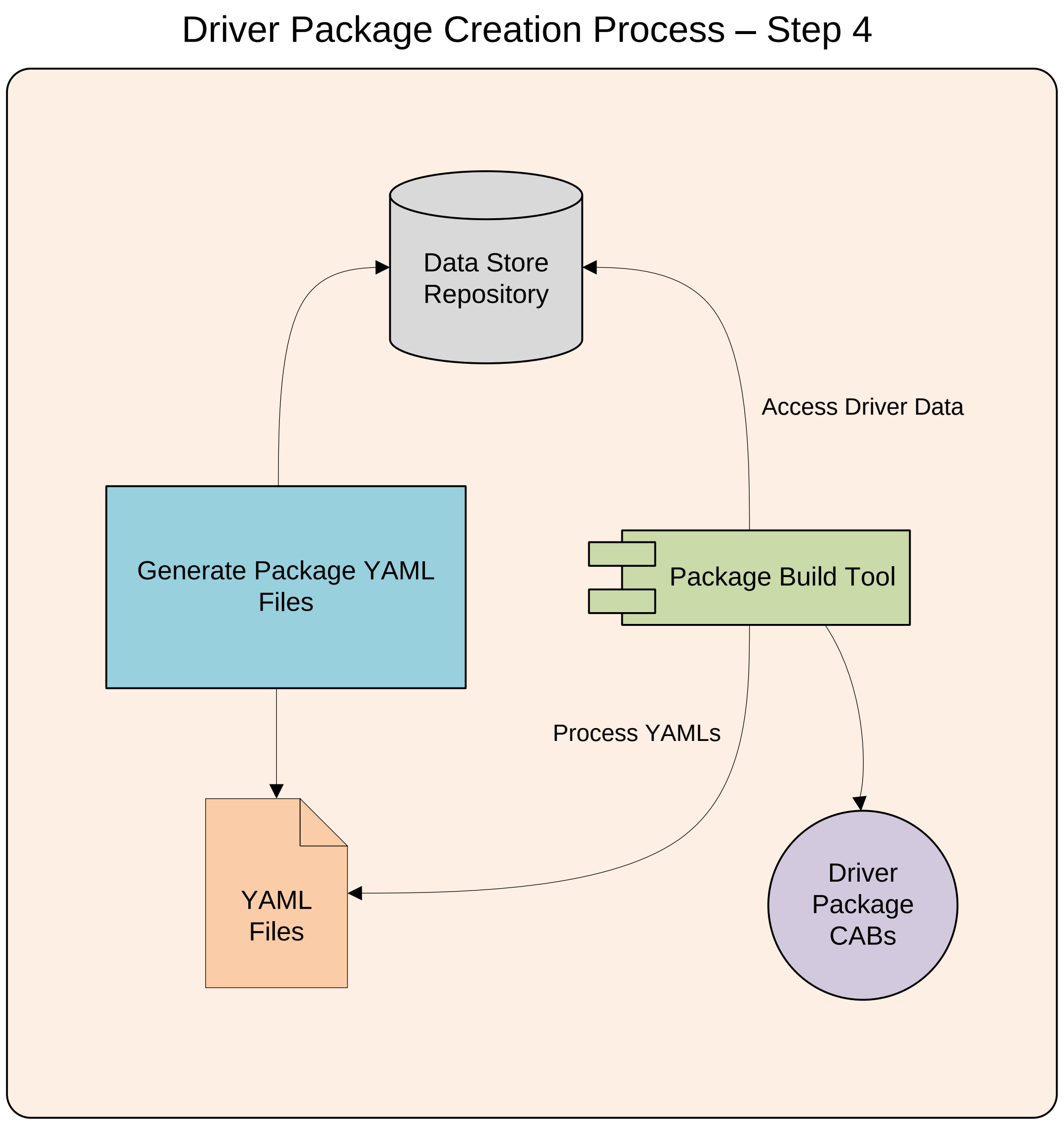
```
%YAML 1.1
---
#
# ---------------------------------------------------------------------------
```

```
name:           $(MWE)INF-hdaudio
displayName:    Microsoft UAA Function Driver for High Definition Audio (HDAudio)
status:         official
restart:        true
category:       Media

dependsOnPackages:
- $(MWE)INF-ks
- $(MWE)INF-wdmaudio

# ---------------------------------------------------------------------------
# package components section

components:
- hdaudio.inf

#----------------------------------------------------------------------------
# eof
...
```

Examine theYAML file closely. Notice that the driver package dependencies are defined as "dependsOnPackages" YAML key values. This particular driver package depends on two packages: "WinEmb-INF-ks" and "WinEmb-INF-wdmaudio". The "category" YAML key value: "Media" is the extracted driver class. The "components" YAML key value: "hdaudio.inf" is the driver component name. This is the same as the driver INF filename. Windows driver components are named the same as the driver INF filename.

Note: we do not have to store the hardware and compatible IDs in the YAML file. This information will be pulled directly from the data store repository during package creation.

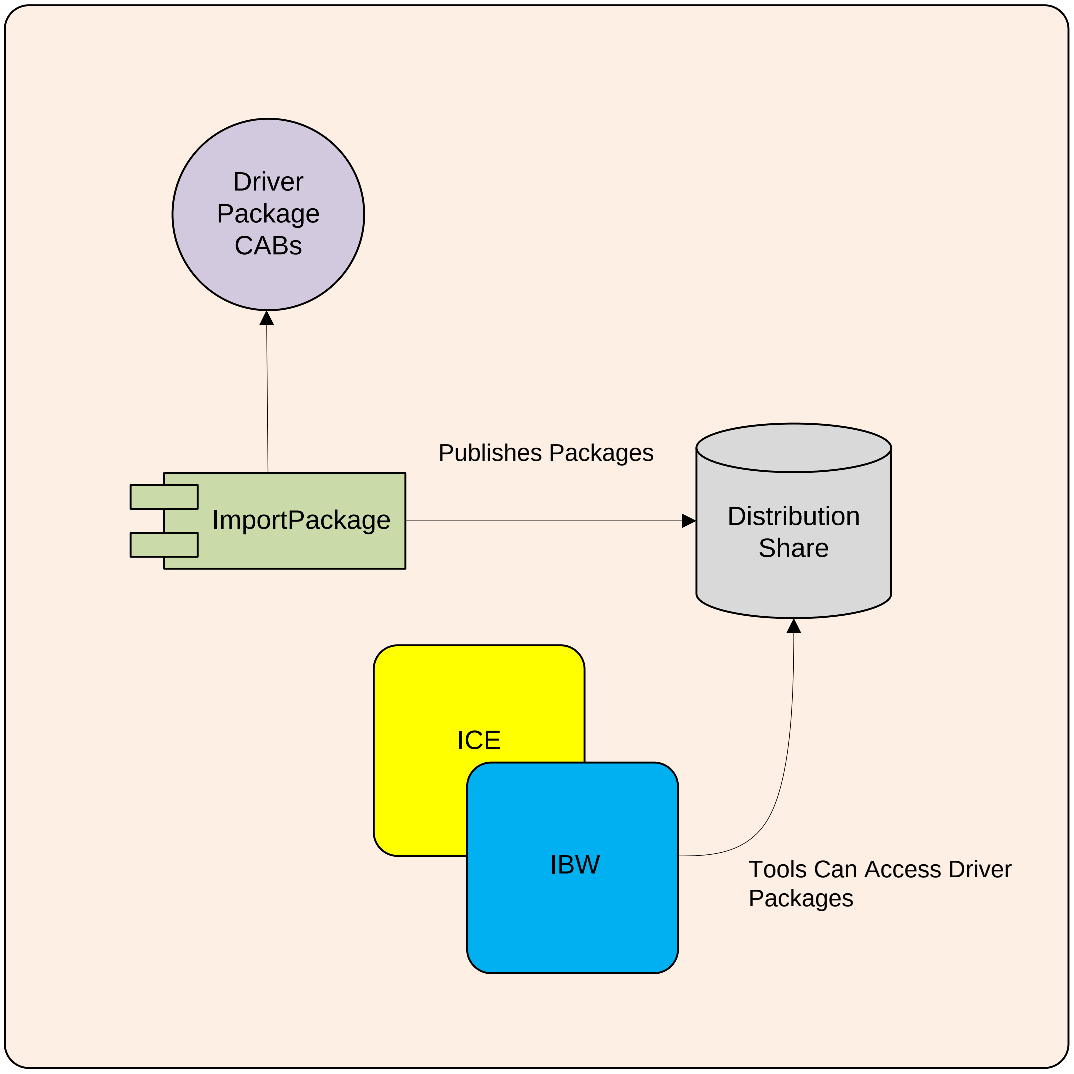
The package build tool processes all the YAML files, while referencing the data repository for associated driver data. Then it outputs package "Cabinet" (CAB) files for each. The final package CAB files are then published. This is described next.



Publishing the Packages

The final package CAB files are published by importing them into the “Distribution Share” (DS). This is a package repository accessed by IBW and ICE. Package CAB files are imported into the DS by invoking the **ImportPackage** tool.

Driver Package Creation Process – Step 5



Device Discovery

As mentioned earlier, you can run TAP.EXE to generate a DEVICES.PMQ file. PMQ files can be imported into IBW or ICE when you build an image to automatically map driver packages from the DS which match the device identification strings in the PMQ. IBW will internally start TAP.EXE to automatically map driver packages, if you select the option to automatically map drivers to devices.

Installing Additional Drivers

After you build an image, and deploy it to an embedded device, you may want to install additional drivers after booting into it. You can do this in ICE by creating a new answer file, and then adding the desired driver packages and associated package dependencies, followed by creating a “configuration set”. This configuration set can then be deployed to an image online, by using the **Deployment Image Servicing and Management tool (DISM)** utility.

Conclusion

The result of all this work gives embedded system designers the ability to fine tune driver installation on an embedded device. This provides great benefits and flexibility for prototyping embedded image deployments and for building optimized production ready images.

If you are interested in how to diagnose driver setup issues, see the following series of blog articles: Diagnosis of Driver Setup Issues in Windows Embedded Standard 7 – part 1, 2 and 3.

Best of luck!