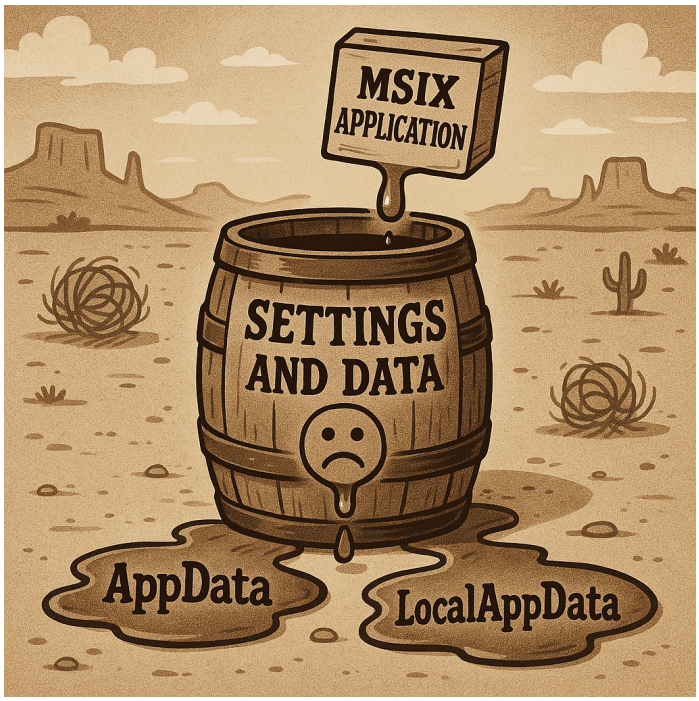
Flexible Virtualization

An under-documented MSIX feature

that we may need!



A Community White Paper by:

Tim Mangan

TMurgent Technologies, LLP

July 1, 2026

Introduction

We are constantly looking for ways to improve the compatibility of running traditional Win32 and Dot Net Framework based applications inside an MSIX container. This often includes adding the Package Support Framework to the MSIX package, but there are other lesser-understood features that were not originally part of MSIX but were added without fanfare or even much documentation.

In 2022 I researched and wrote about two features that we had discovered as extensions to MSIX that were under-documented, *Installed Location Virtualization* and *PackageWriteRedirectionCompatibilityShim*. The results of that research are available for download from https://www.tmurgent.com/appV/en/resources/white-paperlist/597-installed-location-virtualization.

The conclusions of that paper was:

- That ***PackageWriteRedirectionCompabibilityShim*** appeared to be a one-off that Microsoft built for a specific software vendor, and that we were unable to acquire the shim file from Microsoft that was necessary to be able to use the feature.
- That ***Installed Location Virtualization*** (ILV) extension was found to be helpful, although not necessarily as a replacement for the Package Support Framework (PSF), but could be used to augment it. We since made changes to the PSF's MfrFixup to be able to work cooperatively with ILV.MfrFixup in Ilv-aware mode, in conjunction with ILV has proved to be a very effective combination; used in the majority of applications that we repackage into MSIX today.

At the time which that paper was written, we noticed that there seemed to be another possible extension, ***FileSystemWriteVirtualization*** in the new desktop6 schema, but not enough information to be able to use it for effect.

Since that time additional missing components for FileSystemWriteVirtualization have appeared in the Virtualization namespace, however we could not get them to create valid Manifest files. Recently, we discovered that there is an error in the AppXManifest Schema that supports these new extensions, and by correcting the schema used to generate the package we could create such packages. Furthermore, we happily discovered that when installing the package either Microsoft was skipping the validation for these elements, or had quietly fixed the copy of the schema used by the Desktop AppInstaller without releasing this fix.

We have found this technique to be needed in at least one application that fails UAT testing for file access, even with the PSF's MfrFixup in Ilv-aware mode with the ILV. So it is time to investigate just what this feature does and does not do so that we can choose the right solution for each application with some level of criteria that is better than throwing darts against the wall.

Note for short-attention span individuals: There is a short summary section at the end of this document.

Flexible Virtualization

In researching this topic, we located a Microsoft web page on Microsoft Learn with the title "*Flexible Virtualization*", and this page attempts to describe the features we want to investigate and even has an example of how it is to be used. The paper is incomplete, and at times possibly misleading to those reading it by not being careful enough with terms. Still, it makes a good place to start to understand the what and whys and I recommend you give it a once over now. Find this page at https://learn.microsoft.com/en-us/windows/msix/desktop/flexible-virtualization

Flexible Virtualization uses three different schema extensions. Part of the goals of this paper is to understand those items separately and as a group; to determine what combinations are possible and how they might be useful. The other part is to attempt to define exactly what these things do.

The three parts to the schema extensions that fall under Flexible Virtualization are:

- A new rescap:capability setting, unvirtualizedResources.
- desktop6:FileSystemVirtualization and desktop6:RegistryVirtualization
- virtualization:FileSystemVirtualization and virtualization:RegistryKeyVirtualization.

Note that this paper will not look into the Registry Virtualization control portions.

Ultimately, the intent of using Flexible Virtualization is to carve out exclusions for certain file paths such that if the application makes changes or additions to files in that area, they will be written out to the native location rather than be redirected to the MSIX packages folders under %LocalAppData%, presumably because of the need to share those files with other applications in other packages (or native).

Schema Fix

We mentioned a need for a fix to the official Microsoft schema used to validate the AppXManifest file. The error occurs in the VirtualizationManifestSchema.xsd file, which is officially distributed as part of Visual Studio. In addition to defining the syntax for element and parameters, it also defines validation syntaxes. These appear in the form of RegEx[1] patterns. We define the fix here, for any third-party vendor needing it:

The problem with the current schema file is in the RegEx pattern used to validate a ExcludedDirectory string. The incorrect pattern is in the definition on line 61 for the simple type ST_ExcludedDirectory, which includes in part:

```
<xs:pattern value="$\([kK][nN][oO][wW][nN][fF][oO][lL][dD][eE][rR]:[A-Za-z0-
9]{1,32}\)(\\.+)?"/>
```

When it should be:

[1] MSIX schemas use the ECML flavor of Regex, sometimes also referred to as the javascrpt flavor.

<xs:pattern value="\$\([kK][nN][oO][wW][nN][fF][oO][lL][dD][eE][rR]:[A-Za-z0-9]{1,32}\)(\\.+)?"/>

This modification aligns with the text descriptions and examples on Microsoft Learn, which Microsoft tells us is the official documentation for these elements and that the schema files are not. That might not make sense, but we have a fix so let's just move on.

Who Does What Where in the app?

We have discovered that there are two sets of APIs that we have to be concerned with when determining the effect of running traditional software in a MSIX container, with any mix of file based virtualization features such as the The PSF MfrFixup, MSIX Runtime, FileSystemWriteVirtualization, or Installed Location Virtualization.

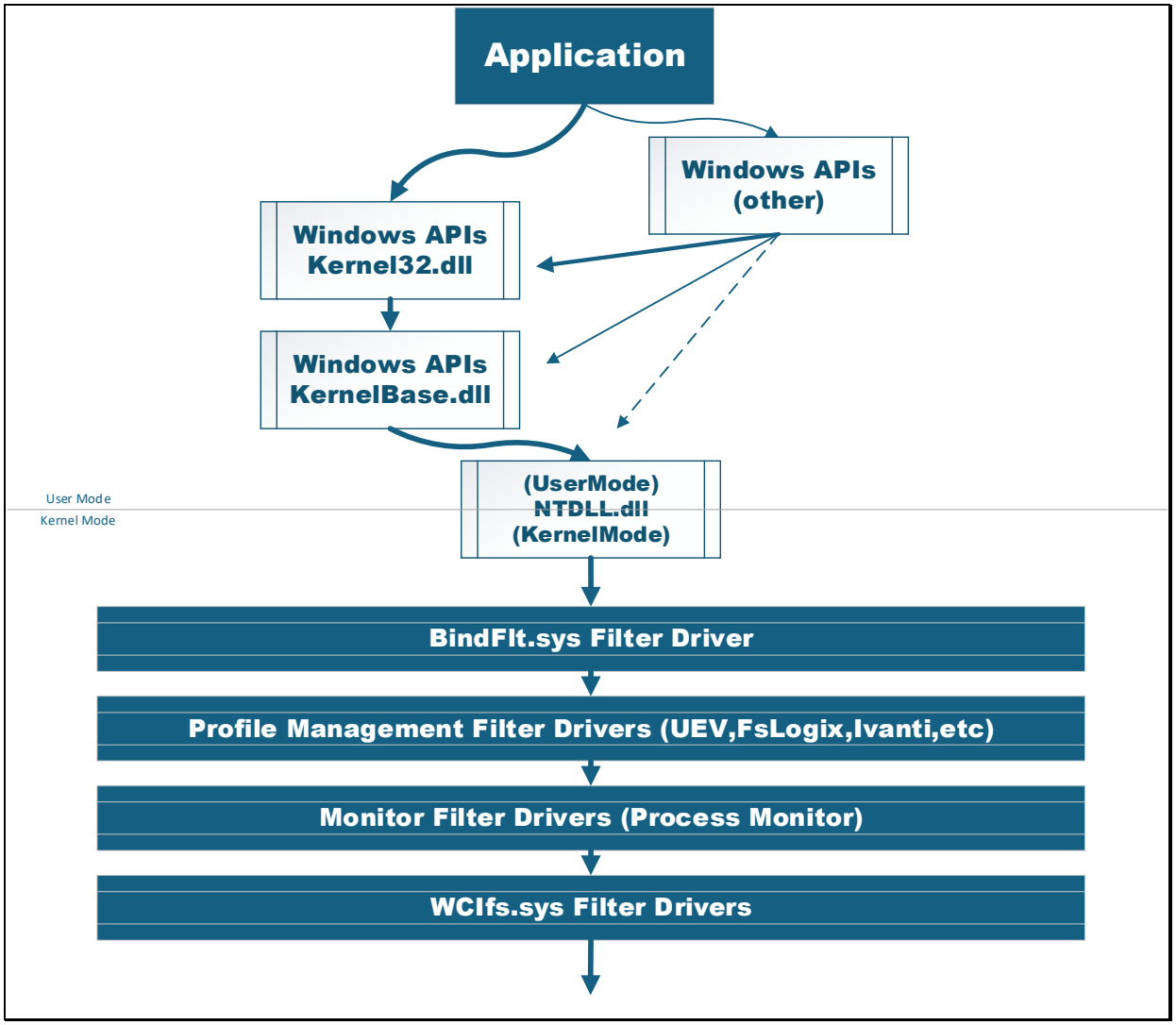
- Traditional Win32 and .Net Framework file access, typically calling standard APIs located in Kernel32.dll, and KernelBase.dll.. This is your basic calls like CreateFile.
- Newer APIs, typically added for UAP originally but now also available for traditional applications to use. This includes using APIs located in System.Storage.dll. Some of those calls still pass through the APIs, but some do not. These newer APIs are not targeted by the PSF, nor ILV (in general), so we tend have more problem running these apps under MSIX today than others, unless the app is written for the newer .Net Runtimes (like .Net 5 and above).

Therefore in testing we had to be quite certain what the application was calling and thus the testing performed here was with special purpose applications to validate different scenarios. For the IT Pro wanting understand these features, we caution that you may need to learn a little bit more about the application to determine the best solution. Also, if you don't get the results you expect in an app after using these features, it is most likely that the app is not using the APIs you thought it was!.

We have also found it useful to divide up the various native locations where apps look for and create files into buckets that are generally treated the same way, no matter which virtualization technologies are in play. The following buckets were considered in the testing used to produce this paper:

Bucket	Examples	Notes
Well known binary paths	Program Files, Program Files (x86), CommonProgramFiles, Windows	
Local AppData	%profile%\AppData\Local	Except Temp sub-folder
Roaming AppData	%profile%\AppData\Roaming	
Common AppData	C:\ProgramData	
Documents/Desktop	The user's documents folder and Desktop	
Elsewhere	C:\PrivateVendorFolder, %profile	Except for local and remote AppData, and Documents/Desktop

The following is a diagram showing the flow of an application file call from the application down into the kernel of the OS to reach the file system with important interfaces/functions shown in a layered mannor. Having this visual should be helpful when reading the details of this paper.



Most file calls follow the heavier line paths, however, certain APIs, like System.Configuration, have been seen to call to different levels in some cases, bypassing the Kernel32/KernelBase layers.

Most of the work of the raw MSIX Runtime is believed to take place in NTDLL, most likely on the kernel side functions. The Package Support Framework targets mostly Kernel32.dll and KernelBase.dll with a small portion of support also in the user side functions of NTDLL. MSIX Extensions appear to be implemented in BindFlt.sys, with some support possibly in the NTDLL places where MSIX runtime support lives.

WCIfs has some support for the virtualized container, but we have not detected anything that affects file calls to date.

What parts of the package files are affected by what?

When we package Win32 and .Net Framework applications into MSIX packages, upon installation of the package all of the files are located in different locations than they would have been with an MSI or EXE installation.

The ability of the application to locate and manipulate those files become affected by the base MSIX runtime implementation, certain schema extensions, and the presence of a PSF file based fixup like MfrFixup. But each of these tend to affect subsets of the files in the package, and in different ways.

MSIX Runtime

While possibly not technically correct, the changes made to application file requests occur in the area of the user-mode to kernel-mode interrupt. For simplicity, we can think of this as living in the ZW functions in kernel side of ntdll.dll.

An example of this is that if the application requests to open a file as "C:\Program Files\Vendor\foo.xyz", this runtime will first check if that file is present in the package folder VFS\ProgramFilesX64\Vendor and if not found then check the requested location.

The MSIX Runtime support for files is part of the basic operating system. Compared to a natively installed application, MSIX will generally provide file mapping from native to packaged locations for the 'Well-known binary paths', and 'Common AppData' buckets only. MSIX does not allow package files to be updated, except for the 'Common AppData' buckets where a special package redirection area is utilized.

The MSIX Runtime helps the application by looking at file requests with an eye to layer package VFS files over the native file path equivalent, but only for a subset of VFS file paths. Specifically, it layers package files from the following VFS folders only:

- ProgramFilesX86
- ProgramFilesX64
- ProgramFilesX86CommonFiles
- ProgramFilesX64CommonFiles
- Windows, as well as variablized names that fall under the C:\Windows folder.
- Common AppData (c:\ProgramData)[2]

When an application runs inside the package, any request using either the native path (such as C:\Program Files\Vendor) OR the package path (such as C:\Program Files\WindowsApps\{PackageFullName}\VFS\ProgramFilesX64\Vendor) will see a view of the package files overlaid on the native files.

The MSIX Runtime does not map other VFS portions of the package. Nor does it allow any package files to be modified. This is why we have the other options!

[2] Originally not included in the MSIX Runtime, support for this mapping was added in 2017.

The MSIX Runtime does, however, have functionality that will detect if the packaged application attempts to write to the user's AppData\Roaming folder, in which case it will redirect that write to a special place in the users AppData\Local\Packages\{PackageFamilyName} area. This redirected area also serves for an overlay to the user's Roaming folder when file discovery/read operations are performed.

Package Support Framework MfrFixup

The Package Support Framework MfrFixup is implemented as an intercept to calls made by the application in User Mode, fitting in-between the application code and Windows API functions implemented in Kernel32.dll, Kernelbase.dll, and sometimes the Nt functions on the user mode side of ntdll.dll.

The PSF MfrFixup aims to overcome many of the limitations imposed by the MSIX Runtime mappings. This includes making package files visible to the application, and allowing most files to be updated. As files may be changed, The MfrFixup deals with three possible layers, native, package, and redirected.

Generally, the MfrFixup allows the app to use any of the layer references to see an overlay of the 3 layers. In practice, we often see apps get "confused" as they open a file that is in a different location than they expected, then query the open handle to determine a path for another file they want to operate on. This confusion causes the app to try the wrong path, but usually MfrFixup will do the right thing.

The MfrFixup also defines two different modes for redirection. One is to redirect to the packages folder mentioned previously, although to a different subfolder, called 'traditional containerized redirection'. The other is to redirect to the native path, called 'local redirection'. Most filepaths are set up for the traditional redirection, but for certain folders (like the user's documents folder) local redirection makes the most sense. Local redirection is important whenever the updated/new file may be needed by software running outside of the container.

By default, the MfrFixup does not allow binary files to be updated, however there is an override available for this. Please don't use this override to try allowing the app to update itself.

The MfrFixup performs these functions as a user-mode injected dll that sits in between the application making the request and the Windows APIs in kernel32.dll or kernelbase.dll (and sometimes the user-mode side of ntdll.dll

InstalledLocationVirtualization

InstalledLocationVirtualization (ILV) is believed to perform these functions in the BindFlt.sys filter driver that lives inside the kernel at the top of the mini-filter driver stack, just below the kernel-mode part of ntdll.dll. There is some evidence that the MSIX Runtime may also be aware of ILV and operate slight differently when in place.

ILV is a package AppXManifest extension that allows a subset of package files to be updated, with redirection to the packages folder.

The files affected by ILV are only for those folders covered by the MSIX Runtime. Also, the application may not receive full layering services if it becomes confused and addresses the redirection area directly, but both native and package paths work great.

The ILV does not allow binary files to be updated.

FileSystemWriteVirtualization

We have not yet determined where FileSystemWriteVirtualization is implemented. FileSystemWriteVirtualization is likely also implemented in BindFlt.sys, and possibly the ZW functions in the kernel mode side of ntdll.dll.

FileSystemWriteVirtualization is a set of package AppXManifest additions that allows a different subset of package files to be updated, with redirection to the native folder. It is targeted towards only files accessed using the native Local and Remote AppData buckets. Typically we target all of this area, but by configuration it can be targeted to a subset of them.

All together now

These different options may be combined, when needed.

Changes in the MfrFixup made when we wrote the ILV white-paper allow MfrFixup to recognize that ILV is in use and defer to ILV when appropriate. This mode must be set in the MfrFixup when ILV is also in use. This combination is now the preferred combination when file help is needed.

As part of the work for this white-paper, we also found it necessary to be able to inform MfrFixup when FileSystemWriteVirtualization is in use, and modify the behavior of MfrFixup. Rather than it being a mode, we now can signal changes in the MfrFixup allow for altering the direction of redirection for the AppData/Local and AppData/Roaming folders from the traditional style of redirection to use the local style of redirection. If both the MfrFixup and FileSystemWriteVirtualization are needed, the MfrFixup configuration should be adjusted to override the redirection style for these folders.

Generally, we first look to add the Psf MfrFixup in Ilv-aware mode with ILV when the package requires file help over the support by the MSIX runtime alone. We also generally want the AppData/Local and AppData/Remote changes redirected using the traditional style redirection, so we only use FileSystemWriteVirtualization when needed. If file help outside the bounds of AppData folders are needed, the Psf MfrFixup and ILV may be combined with it.

Testing Results

Testing was performed using a purpose-built application that would conduct certain exact API calls on demand, making the process of detecting the results to the app, and impact to the filesystem certain.

Review of ILV paper results

These are the results from the ILV paper for review, which will be helpful in understanding the new testing results that will follow. All results assume that the file in question exists in the package.

R=Open file for read
W=Open file for write
D=Delete file

File Path Requested by app	MSIX	MSIX+ILV	MSIX_MFR	MSIX+ILV+MFR
C:\Program Files\Vendor	R W D	R W[3] D	R W[3] D[4]	R W[3] D[4]
<Root>\VFS\ProgramFilesX64\Vendor	R W D	R W[3] D	R W[3] D[4]	R W[3] D[4]
%localappdata%\Vendor	R W D	R W D	R W[5] D[4]	R W[5] D[4]
<Root>\VFS\Local AppData\Vendor	R W D	R W[5] D	R W[5] D[4]	R W[5] D[4]
%appdata\Vendor	R W D	R W D	R W[6] D[4]	R W[6] D[4]
<Root>\VFS\AppData\Vendor	R W D	R W[6] D	R W[6] D[4]	R W[6] D[4]
Modern "System.Configuration"[7]	R W	R W	R W	R W

In that testing, we did not consider the Common AppData bucket, but we do now cover this in the new testing that will follow.

We concluded that when file help is needed for an MSIX package, Msix+Mfr+Ilv is the first and best choice to make. Nothing in this paper changes that finding, but we have options when something different is required.

[3] Redirected to Packages\XX\LocalCache\Microsoft\WritablePackageRoot\VFS\ProgramFilesX64\Vendor
[4] The MFR will fail to delete the file, but return a successful response for app compatibility.
[5] Redirected to Packages\XX\LocalCache\Microsoft\WritablePackageRoot\VFS\LocalAppData\Vendor
[6] Redirected to Packages\XX\LocalCache\Microsoft\WritablePackageRoot\VFS\AppData\Vendor
[7] App that uses System.Configuration API to store configuration information. Failure to write often leads to app crash or reinstall prompts.

New Testing Results (for traditional APIs)

The new testing adds in the Common AppData bucket, as well as the Documents bucket. It also adds in the most common scenario for the app to add new files. Location of the redirected files is also confirmed.

The testing was performed with an PSF updated in late June 2025 for changes mentioned in this paper to MfrFixup, as well as a pre-release version of TMEditX 6.0.
An example of part of the AppXManifest file, showing the Flexible Virtualization and InstalledLocationVirtualization changes needed for some of the packages is shown here:

```
<Package xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10" xmlns:uap="http://schemas.microsoft.com/app
x/manifest/uap/windows10" xmlns:uap3="http://schemas.microsoft.com/appx/manifest/uap/windows10/3" xmlns:uap10="http://schem
as.microsoft.com/appx/manifest/uap/windows10/10" xmlns:desktop6="http://schemas.microsoft.com/appx/manifest/desktop/windo
ws10/6" xmlns:rescap="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabilities" xmlns:virtualizatio
n="http://schemas.microsoft.com/appx/manifest/virtualization/windows10" IgnorableNamespaces="desktop6 rescap uap uap10 uap3
virtualization">
<Identity Name="TMurgent-BadAssTest" Publisher="CN=TMurgent.local" Version="3.0.0.3" ProcessorArchitecture="x64"/>
<Properties>
   <DisplayName>TMurgent BadAssTest</DisplayName>
   <PublisherDisplayName>Packaged by TMurgent Technologies, LLP</PublisherDisplayName>
   <desktop6:FileSystemWriteVirtualization>disabled</desktop6:FileSystemWriteVirtualization>
  <uap10:PackageIntegrity> <uap10:Content Enforcement="on"/> </uap10:PackageIntegrity>
  <virtualization:FileSystemWriteVirtualization>
    <virtualization:ExcludedDirectories>
      <virtualization:ExcludedDirectory>$(KnownFolder:LocalAppData)</virtualization:ExcludedDirectory>
      <virtualization:ExcludedDirectory>$(KnownFolder:RoamingAppData)</virtualization:ExcludedDirectory>
  </virtualization:ExcludedDirectories>
</virtualization:FileSystemWriteVirtualization>
</Properties>
<Resources>
  <Resource Language="en-us"/>
</Resources>
<Dependencies>
  <TargetDeviceFamily Name="Windows.Desktop" MinVersion="10.0.19041.0" MaxVersionTested="10.0.22635.1"/>
</Dependencies>
<Capabilities>
  <rescap:Capability Name="runFullTrust"/>
  <rescap:Capability Name="unvirtualizedResources"/>
</Capabilities>
<Applications>
<Application Id="TMURGENTBADASSTESTPSFLAUNCHEROne" Executable="TMurgent-
BadAssTest_PsfLauncher1.exe" EntryPoint="Windows.FullTrustApplication">
<uap3:VisualElements BackgroundColor="transparent" DisplayName="BadAssTest" Square150x150Logo="Assets\Square150x150Logo.png
" Square44x44Logo="Assets\Square44x44Logo.png" Description="BadAssTest" VisualGroup="BadAssTest">
<uap:DefaultTile Wide310x150Logo="Assets\Wide310x150Logo.png" Square310x310Logo="Assets\Square310x310Logo.png" Square71x7
1Logo="Assets\Square71x71Logo.png"/>
</uap3:VisualElements>
</Application>
</Applications>
<Extensions>
  <uap10:Extension Category="windows.installedLocationVirtualization">
  <uap10:InstalledLocationVirtualization>
    <uap10:UpdateActions ModifiedItems="keep" AddedItems="keep" DeletedItems="keep"/>
  </uap10:InstalledLocationVirtualization>
</uap10:Extension>
</Extensions>
</Package>
```

The changes to the config.json file to support MfrFixup being Flexible Virtualization aware, when needed, used in this testing is shown here:

```
…
"fixups": [
      {
       "dll": "MFRFixup.dll",
       "config": {
        "ilvAware": "true",
        "overrideCOW": "default",
        "overrideTraditionalRedirections": [
         {
          "name": "Local AppData",
          "mode": "local"
         },
         {
          "name": "AppData",
          "mode": "local"
         }
        ]
       }
      },
…
```

In the following subsections, detail is provided on testing results with various combinations of file assistance help.

The tests include reading and writing to files that exist in the package, but using various paths that the application may use. The “read” items test not only by direct request by name, but the ability to enumerate a directory as a layered collection is validated. The “delete” items indicates the app requesting to delete the file. The “new” items test the ability to create a new file using a package like path. Only likely scenarios are tested for “new” items. When a particular test was not verified as part of the testing, it is left blank, or there may be an entry result in parentheses indicating what we believe would happen.

In scenarios where redirection occurs, there are apps that utilize the actual location of a file after redirection to create the path for a different file request. For those situations testing of calls using the redirection path is included; in scenarios where that would not occur the testing is not performed and is marked N/A.

Raw MSIX

This refers to a package as created by the Microsoft MSIX Packaging Tool without additional manipulation. The results are shown in the table below:

Scenario:	RAW MSIX		Read	Write	Delete	New
Bucket:	Program Files	Native	Yes	No	No	(no)
		Package	Yes	No	No	No
		PkgRedir	N/A	N/A	N/A	N/A
	Local AppData	Native	No	No	No	LC/L
		Package	Yes	No	No	N/A
		PkgRedir	N/A	N/A	N/A	N/A
	Remote AppData	Native	No	No	No	LC/R
		Package	Yes	No	No	
		PkgRedir	N/A	N/A	N/A	N/A
	Common AppData	Native	Yes	No	No	Native
		Package	Yes	No	No	
		PkgRedir	N/A	N/A	N/A	N/A
	C:\Private	Native	No	No	No	(native/no)
		Package	Yes	No	No	
		PkgRedir	N/A	N/A	N/A	N/A
	Documents/Personal	Native	No	No	No	(yes)
		Package	Yes	No	No	(no?)
		PkgRedir	N/A	N/A	N/A	N/A

The Native/Package/PkgRedir column refers to the bucket actually requested by the application. We find that under MSIX, some applications will make some calls to the Package or PkgRedir locations in addition to native paths. This occurs because either the change in process working directory, or the app requesting the path to an open handle of a different file it has open and offsetting from those paths when asking to work with a different file.

LC/L and LC/R refer to special folders in the redirection area that are not part of the WritablePackageRoot. This is only of significance to external profile management solutions looking to roam these files.

The results in parentheses were not actually tested but we believe these to be what we would have seen. Blank entries were not tested as unlikely to occur. N/A is short for Not Applicable as the scenario would not occur.

UDF

UDF is shorthand for adding in the three parts to Flexible Virtualization, the capabilities change, the desktop6 change, and the virtualization change where all of the local and remote AppData folders are placed in the virtualization exclusion list. In the test results shown below, this was the only change made to the package from the MSIX Packaging Tool.

Scenario:	UDF		Read	Write	Delete	New
Bucket:	Program Files	Native	Yes	No	No	(no)
		Package	Yes	No	No	No
		PkgRedir	N/A	N/A	N/A	N/A
	Local AppData	Native	No	No	No	**Native**
		Package	Yes	No	No	
		PkgRedir	N/A	N/A	N/A	N/A
	Remote AppData	Native	No	No	No	**Native**
		Package	Yes	No	No	
		PkgRedir	N/A	N/A	N/A	N/A
	Common AppData	Native	Yes	No	No	Native
		Package	Yes	No	No	
		PkgRedir	N/A	N/A	N/A	N/A
	C:\Private	Native	No	No	No	(native/no)
		Package	Yes	No	No	
		PkgRedir	N/A	N/A	N/A	N/A
	Documents/Personal	Native	No	No	No	(yes)
		Package	Yes	No	No	(no?)
		PkgRedir	N/A	N/A	N/A	N/A

The **Bold** items highlight differences from the RAW package.

While some testing was done on sub-components of U D and F separately, we found no value in using a subset (only problems).

UDF+ILV

UDF + ILV package was made like the UDF package, but we also added InstalledLocationVirtualization.

Scenario:	UDF + Ilv		Read	Write	Delete	New
Bucket:	Program Files	Native	Yes	No	No	
		Package	Yes	**Yes**	**Clean**	**PkgRedir**
		PkgRedir	**Yes**	**Yes**	**Clean**	
	Local AppData	Native	No	No	No	**Native**
		Package	Yes	**Yes**	**Yes**	
		PkgRedir	**Yes**	**Yes**	**Yes**	
	Remote AppData	Native	No	No	No	**Native**
		Package	Yes	**Yes**	**Yes**	
		PkgRedir	**Yes**	**Yes**	**Yes**	
	Common AppData	Native	Yes	No	No	Native
		Package	Yes	**Yes**	**Yes**	
		PkgRedir	**Yes**	**Yes**	**Yes**	
	C:\Private	Native	No	No	No	(native/no)
		Package	Yes	**Yes**	**Clean**	
		PkgRedir	**Yes**	**Yes**	**Clean**	
	Documents/Personal	Native	No	No	No	(yes)
		Package	Yes	**Yes**		**(pkgRedir)**
		PkgRedir				

The **Bold** items highlight differences from the RAW package.

The items with **thicker Boxing** highlight differences from the UDF only scenario.

Clean indicates a partial or full deletion occurred. A file in the package automatically creates a back-link placeholder in the Package Redirection area (WritablePackageRoot). That back-link may be overwritten by a write action, and new files may also be added in that rediretion area. A delete of a package file in this case, deletes the back-link redirection file, and the application does not see an error. However, if there was a file in the package, it still exists and subsequent actions by the app will find the original file. As apps usually don't try to delete package files, but only added files in the same area, this should not cause an issue.

Mfr+Ilv

In this scenario, the RAW package was updated using the Psf, including the MfrFixup in Ilv-aware mode, and ILV.

Scenario:	Mfr + Ilv		Read	Write	Delete	New
Bucket:	Program Files	Native	Yes	Yes	Clean	
		Package	Yes	Yes	Clean	PkgRedir
		PkgRedir	Yes	Yes	Clean	
	Local AppData	Native	Yes	Yes	Clean	PkgRedir
		Package	Yes	Yes	Clean	
		PkgRedir	Yes	Yes	Clean	
	Remote AppData	Native	Yes	Yes	Clean	PkgRedir
		Package	Yes	Yes	Clean	
		PkgRedir	Yes	Yes	Clean	
	Common AppData	Native	Yes	Yes	Clean	PkgRedir
		Package	Yes	Yes	Clean	
		PkgRedir	Yes	Yes	Clean	
	C:\Private	Native	Yes	Yes	Clean	
		Package	Yes	(yes)	(clean)	
		PkgRedir	(yes)	(yes)	(clean)	
	Documents/Personal	Native	Yes	(yes)		
		Package	Yes	Yes		
		PkgRedir				

The **Bold** items highlight differences from the RAW package.

Clean is as defined in the previous scenario.

In the Private bucket, redirected files also go to the PkgRedir area (WritablePackageRoot).

In the Documents/Personal bucker, redirected files go to native locations.

Mfr + IlV + UDF (Full Solution)

In this scenario, the RAW package had the Psf MfrFixup in Ilv-aware mode added, ILV, added, and UDF added.

Scenario:	Full Fix		Read	Write	Delete	New
Bucket:	Program Files	Native	Yes	Yes	Clean	
		Package	Yes	Yes	Clean	PkgRedir
		PkgRedir	Yes	Yes	Clean	
	Local AppData	Native	Yes	Yes	CleanN	Native
		Package	Yes	Yes	CleanN	
		PkgRedir	Yes	Yes	CleanN	
	Remote AppData	Native	Yes	Yes	CleanN	Native
		Package	Yes	Yes	CleanN	
		PkgRedir	Yes	Yes	CleanN	
	Common AppData	Native	Yes	Yes	Clean	PkgRedir
		Package	Yes	Yes	Clean	
		PkgRedir	Yes	Yes	Clean	
	C:\Private	Native	Yes	Yes	Clean	
		Package	Yes	(yes)	(clean)	
		PkgRedir	(yes)	(yes)	(clean)	
	Documents/Personal	Native	Yes	(yes)		
		Package	Yes	Yes		
		PkgRedir				

The **Bold** items highlight differences from the RAW package.

The items with **thicker Boxing** highlight differences from the Mfr+Ilv only scenario.

Clean is defined as in the prior scenario.

CleanN means that the native file (if present) is removed, but if there was a package file by the name it is not removed.

With this combination, no matter how the app tries to open/create the file under AppData or Local AppData, MfrFixup will redirect that call back to the native path in user mode, and then flexible virtualization will kick in to override the normal MSIX runtime behavior.

- *If the file exists natively, it will use that.*
- *If not, and the file exists in the package, it will copy-on-write to the native path and use it there.*
- *If not also in the package, it will create a new file in the native path (or return file not found if appropriate for the call)..*

New Testing Results (for System.Configuration)

Some applications handle application settings using a newer API that was originally developed for UWP apps, but then extended to allow older apps to use it.

In this API, the application isn't supposed to know where the configuration is stored, but may request the settings to be:

- Application/User Level
- Local
- Roaming

The vendor provides the *application/User level* settings in an xml file that is placed next to the application exe. If the exe was named foo.exe then this file would be named foo.exe.config. This config file includes other items, so we cannot assume there are application level configuration settings from the presence of this file alone. What makes these setting application or user level is specified by how the application calls the API to get access to them. Application level is intended to be read-only, and User-level is intended to be read-write, however as the file is likely to be placed in a read only area (Program Files), the write will fail (unless something else redirects it behind the scenes). So apps should not use User Level settings for read/write, but one of the other supported APIs.

The other two selections, would add a folder to the user's appdata/local or appdata/remote folder and place the settings in a file under there. In a native deployment, the API uses a subset of the application name (without the exe, but limited 25 characters) and appends this with "_Url" and a hash for this folder name. Under this folder is creates a folder of the application version string, and places a file called 'user.config' there.

File Path Requested by app	MSIX	Mfr+Ilv	Flex	Flex+Ilv	Mfr+Ilv+Flex
Modern "System.Configuration" (None)	R W	R W[8]	R W[9]	R W[8]	R W[8]
Modern "System.Configuration" (Local)	R W[10]	R W[11]	R W Error! Bookmark not defined.	R W Error! Bookmark not defined.	R W Error! Bookmark not defined.
Modern "System.Configuration" (Roaming)	R W[12]	R W[13]	R W Error!	R W Error! Bookmark not defined.	R W Error! Bookmark not defined.

[8] Redirected to WritablePackageRoot\VFS\ProgramFilesX64\folder
[9] App may crash after attempting to save changes
[10] Redirected to Packages\Pkg\LoalCache\Local\appnam\appnam_Url_hash\version\user.config
[11] Redirected to WritablePackageRoot\VFS\Local AppData\Appnam\Appnam_hash\version\user.config
[12] Redirected to Packages\Pkg\LocalCache\ Roaming\appnam\appnam_Url_hash\version\user.config
[13] Redirected to WritablePackageRoot\VFS\AppData\Appnam\Appnam_hash\version\user.config

			Bookmark not defined.		

As long as the application does not attempt to directly manipulate the actual files, in addition to using the API, we find that there is no problem with the application under any of these scenarios different than that of the native situation.

Summary

Flexible Virtualization offers an additional arrow for our quiver, but in my opinion, one that should be used sparingly.

It provides the ability to “unvirtualize” files modified or created by the application for the AppData or LocalAppData areas, for the purpose of making those files available outside of the package, typically for access by a different application. Another way of stating this is that the files can poke out of the container.

Adding this feature into the package by itself does not allow for “pre-configuration” of the application by having files in the VFS\AppData or VFS Local AppData folders, but combining this with the PSF MfrFixup, we can. This required modifications to MfrFixup to achieve, so you might need to update tooling to get access to it.

An organization implementing MSIX might look at the information provided from this paper and be inclined to choose the full Mfr+Ilv+UDF solution for all of their packages. **I do not recommend this as a general packaging practice for MSIX packages.**

There is great value in ensuring that all of the application data is removed when the package is removed. This not only affects system cleanliness, it will lead to problems should the package be removed (removing some but not all of the app configuration) and a replacement package added.

Use of FlexibleVirtualization should be reserved only for:

- Packages that produce files where those files need to be shared outside of the package (or shared package container).
- Applications that don’t work any other way.