PowerShell; Generate real Excel XLSX files without Excel

I had the need to store data's into a Microsoft Excel compatible file.

The first attempt is to use the Excel COM object model.
This is not a good solution because:
PowerShell runs very often on Servers or clients without a Microsoft Office / Excel installation.
The use of the Excel COM Object can cause errors inside a Scheduled Task.

Excel can read and store CSV data.
The second attempt is to use CSV data (with Export-CSV)
This is even not a good solution because:
CSV is not just another type of Excel file. On opening a CSV data file, Microsoft Excel converts data automatically. This is not acceptable.
If Microsoft Excel outputs an Excel worksheet into a CSV file, the output does not always follow the CSV format rules. Excel only places quotes around certain fields not on all fields. This leads to unreadable CSV files.

I had the following requirements:

- Solution that works in PowerShell 2.0 and 3.0 (and later)
- Create Excel compatible file without having Excel
  (do not use the Excel COM object model)
- Solution which works without 3th party tools
- Should work similar like the Export-CSV Cmdlet
- Should have possibility to append a worksheet with data (-append parameter)

My Internet research shows no solution which fits these requirements.
But I found a C# code to do the Job. So here is my Translation of this code to PowerShell

For C# code see here:
How to use the Office XML file format and the packaging components from the .NET Framework 3.0 to create a simple Excel 2007 workbook or a simple Word 2007 document
http://support.microsoft.com/kb/931866/en-us

The Excel XLSX file format

Starting with the Microsoft Office Version of 2007 Microsoft has changed the default application file formats from old, proprietary, closed formats (DOC, XLS, PPT) to new, open and standardized Open XML formats (DOCX, XLSX and PPTX).
The Office Open XML (also informally known as OOXML or OpenXML) is a zipped, XML-based file format. To represent spreadsheets, charts, presentations and word processing documents.

Office Open XML is standardized by the European Computer Manufacturers Association (ECMA) where they became ECMA-376 and, in later versions, by ISO and IEC (as ISO/IEC 29500).

Every Open XML file is a zip file (a package) typical containing a number of UTF-8 encoded XML files ("parts").

Inside the XML parts of the package Multipurpose Internet Mail Extensions (MIME) types and Namespaces are used as metadata.
The XML parts (files) of the package are encoded in specialized markup languages. In case of Microsoft Excel this is the markup language called SpreadsheetML.
The package also contains relationship files (part). The relationship parts have the extension .rels. They can be found in a folder with the name _rels.
The relationship parts define the relationships between the parts inside the package (internal) and to resources outside of the package (external).
The package may also contain other (binary) media files such as sounds or images.

The structure of the package is organized according to the Open Packaging Conventions as outlined in the OOXML standard.

You can look at the file structure and the files that comprise a XLSX file by simply unzipping the .xlsx file.

.NET classes to create real Excel XLSX file from scratch

With .Net 3.0 Microsoft has introduced the System.IO.Packaging namespace which lives inside the WindowsBase.dll
WindowsBase.dll is one of the core Assemblies used for Windows Presentation Foundation WPF.
(The Windows Presentation Foundation WPF is Microsoft's next generation UI framework to create applications with a rich user experience even for the new Windows 8 tiles GUI.)
So you don't have to worry that WindowsBase.dll moves around or goes away.
WindowsBase.dll can be found in:
C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\WindowsBase.dll

The System.IO.Packaging namespace provides classes that support Office Open XML Zip compressed containers and other formats, that store multiple data objects in a single container.
System.IO.Packaging contains the ZipPackage class to work with Zip compressed package files.

See the Microsoft developer network (MSDN) documentation for this namespace and classes.
http://msdn.microsoft.com/en-US/library/System.IO.Packaging.aspx

PowerShell can use this .NET namespace and can easily deal with XML files, so here is the way to go.

PowerShell code to load the WindowsBase.dll assembly:

```
$Null = [Reflection.Assembly]::LoadWithPartialName("WindowsBase")
```

Anatomy of a minimal Excel XLSX package file

The number and types of the XLSX package parts will vary based on what is in the spreadsheet. I will describe the minimal XLSX needs here:

Minimal package structure

Example of a minimal basic structure, of a XLSX package file, with 1 mandatory worksheet:

./[Content_Types].xml
./_rels/.rels

./xl/workbook.xml
./xl/_rels/workbook.xml.rels
./xl/worksheets/sheet1.xml

Minimal package parts

Required is the main file: [Content_Types].xml

- Required part for all Open XML documents
- Three content types must be defined:
    1. SpreadsheetML main document (for the start part)
    2. Worksheet
    3. Package relationships (for the required relationships)

The [Content_Types].xml part (file) is generated automatically by the ZipPackage class on creation of the Excel XLSX package file.

Here is the PowerShell code to create the package file on disk:

```
# create the main package on disk with filemode create
$exPkg = [System.IO.Packaging.Package]::Open "C:\test.xlsx",
[System.IO.FileMode]::Create)
```

The [Content_Types].xml file contains definitions of the content types included in the ZIP package, such as the main document, the document theme, and the file properties. This file also stores definitions of the file extensions used in the ZIP package, such as the file formats like .png or .wav. So you can store pictures or sounds inside a document.

Example of a minimal [Content_Types].xml part the package contains a workbook with one worksheet:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Types xmlns="http://schemas.openxmlformats.org/package/2006/content-types">
  <Default Extension="bin" ContentType="application/vnd.openxmlformats-
officedocument.spreadsheetml.printerSettings" />
  <Default Extension="rels" ContentType="application/vnd.openxmlformats-
package.relationships+xml" />
  <Default Extension="xml" ContentType="application/xml" />
  <Override PartName="/xl/workbook.xml"
ContentType="application/vnd.openxmlformats-
officedocument.spreadsheetml.sheet.main+xml" />
  <Override PartName="/xl/worksheets/sheet1.xml"
ContentType="application/vnd.openxmlformats-
officedocument.spreadsheetml.worksheet+xml" />
</Types>
```

Required the document “start part”: workbook.xml

- workbook.xml requires one relationship part workbook.xml.rels which links mainly to the worksheets

Example of a minimal workbook.xml part:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<workbook xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships">
  <sheets>
    <sheet name="Table0" sheetId="1" r:id="rId1" />
  </sheets>
</workbook>
```

I use the .NET XML classes to create the XML document part from scratch:

```
# create the Workbook.xml part XML document

# create empty XML Document
$xl_Workbook_xml = New-Object System.Xml.XmlDocument

# Obtain a reference to the root node, and then add the XML declaration.
$XmlDeclaration = $xl_Workbook_xml.CreateXmlDeclaration("1.0", "UTF-8", "yes")
$Null = $xl_Workbook_xml.InsertBefore($XmlDeclaration,
$xl_Workbook_xml.DocumentElement)

# Create and append the workbook node to the document.
$workBookElement = $xl_Workbook_xml.CreateElement("workbook")
# add the office open xml namespaces to the XML document
$Null = $workBookElement.SetAttribute("xmlns",
"http://schemas.openxmlformats.org/spreadsheetml/2006/main")
$Null = $workBookElement.SetAttribute("xmlns:r",
"http://schemas.openxmlformats.org/officeDocument/2006/relationships")
$Null = $xl_Workbook_xml.AppendChild($workBookElement)

# Create and append the sheets node to the workBook node.
$Null =
$xl_Workbook_xml.DocumentElement.AppendChild($xl_Workbook_xml.CreateElement("sheet
s"))
```

The URI is defined as a relative path to the package root. The URI defines the part and the folder(s) to create.
The Namespace in the Create() method declares the type of relationship being defined from the applicable Office Open XML schema.
The GetStream() Method returns the destination file stream to write the XML document.

```
# create the workbook.xml package part

# create URI for workbook.xml package part
$Uri_xl_workbook_xml = New-Object System.Uri -ArgumentList ("/xl/workbook.xml",
[System.UriKind]::Relative)
# create workbook.xml part
$Part_xl_workbook_xml = $exPkg.CreatePart($Uri_xl_workbook_xml,
"application/vnd.openxmlformats-officedocument.spreadsheetml.sheet.main+xml")
# get writeable stream from workbook.xml part
$dest =
$part_xl_workbook_xml.GetStream([System.IO.FileMode]::Create,[System.IO.FileAccess
]::Write)
# write workbook.xml XML document to part stream
$xl_workbook_xml.Save($dest)
```

Required: one (main) relationship part: .rels

- Must be in a _rels folder

After you have created the Workbook.xml part, you have to create the relationship from the Main [Content_Types].xml to the document body Workbook.xml.

The .rels file in the _rels folders is the main top level relationship file in an Office Open XML package file. This file defines relationships between core files in the ZIP package and the applicable Office Open XML schema.

The main relationship file ".rels" and its folder "_rels" is automatically created by a call to the CreateRelationship() Method from the ZipPackage class.

The Target of a relationship is the location of the referenced file. The target can be within the XLSX ZIP package (internal) or outside (external) of the XLSX ZIP package. We store all files and information's inside the ZIP package, so we use the Target mode Internal.

The Namespace declares the type of relationship being defined from the applicable Office Open XML schema. In this case, the file workbook.xml is being defined as type officeDocument. This information tells Excel that the file workbook.xml contains the document body.

The Relationship Id (rId1 in this case) simply provides a unique identifier for the referenced file.

PowerShell code to create the relationship between the package parts [Content_Types].xml and the main document workbook.xml

```
# create package general main relationships
$Null = $exPkg.CreateRelationship($Uri_xl_workbook_xml,
[System.IO.Packaging.TargetMode]::Internal,
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/officeDocumen
t", "rId1")
```

Required one worksheet: sheet1.xml

- Inside the worksheet XML part, the <sheetdata> node is required, but may be empty

Example of a minimal worksheet XML part:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<worksheet xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships">
  <sheetData />
</worksheet>
```

I use the .NET XML classes to create the XML document part from scratch. The Name of the worksheet part used in the URI, is dynamically generated with the pattern Sheet + number + .xml in the $NewWorkSheetPartName variable. (Example names: Sheet1.xml, Sheet2.xml, Sheet3.xml and so on …)

```
# create worksheet XML document

# create empty XML Document
$New_Worksheet_xml = New-Object System.Xml.XmlDocument
```

```
# obtain a reference to the root node, and then add the XML declaration.
$XmlDeclaration = $New_Worksheet_xml.CreateXmlDeclaration("1.0", "UTF-8", "yes")
$Null = $New_Worksheet_xml.InsertBefore($XmlDeclaration,
$New_Worksheet_xml.DocumentElement)

# create and append the worksheet node to the document.
$workSheetElement = $New_Worksheet_xml.CreateElement("worksheet")
# add the Excel related office open xml namespaces to the XML document
$Null = $workSheetElement.SetAttribute("xmlns",
"http://schemas.openxmlformats.org/spreadsheetml/2006/main")
$Null = $workSheetElement.SetAttribute("xmlns:r",
"http://schemas.openxmlformats.org/officeDocument/2006/relationships")
$Null = $New_Worksheet_xml.AppendChild($workSheetElement)

# create and append the sheetData node to the worksheet node.
$Null =
$New_Worksheet_xml.DocumentElement.AppendChild($New_Worksheet_xml.CreateElement("s
heetData"))
```

The URI is defined as a relative path to the package root. The URI defines the part and the folder(s) to create.
The Namespace in the Create() method declares the type of relationship being defined from the applicable Office Open XML schema.
The GetStream() Method returns the destination file stream to write the XML document.

```
# create the worksheet package part

# create URI for worksheet package part
$Uri_xl_worksheets_sheet_xml = New-Object System.Uri -ArgumentList
("/xl/worksheets/$NewWorkSheetPartName", [System.UriKind]::Relative)
# create worksheet part
$Part_xl_worksheets_sheet_xml = $exPkg.CreatePart($Uri_xl_worksheets_sheet_xml,
"application/vnd.openxmlformats-officedocument.spreadsheetml.worksheet+xml")
# get writeable stream from part
$dest =
$part_xl_worksheets_sheet_xml.GetStream([System.IO.FileMode]::Create,[System.IO.Fi
leAccess]::Write)
# write $New_Worksheet_xml XML document to part stream
$New_Worksheet_xml.Save($dest)
```

Required: workbook relationship part

Every folder in a XLSX ZIP package can contain his own _rels folder to define relationships within that folder. The main document folder "xl" always contains a "_rels" folder with relationship parts.
The relationship part for the workbook.xml is named workbook.xml.rels.

The workbook.xml.rels part is created by use of the CreateRelationship() Method from the ZipPackage class, the "_rels" Folder which contains this part is created automatically by use of the URI.
So first you have to create the XML package part files and then you can create the relationships between them.

The unique ID of the relationship is determined from the workbook.xml and dynamically generated with the pattern rID + Number in the variable $NewWorkBookRelId (Example: rID1, rID2, rID3 and so on …).

```
# create workbook to worksheet relationship
$Null = $WorkBookPart.CreateRelationship($Uri_xl_worksheets_sheet_xml,
[System.IO.Packaging.TargetMode]::Internal,
"http://schemas.openxmlformats.org/officeDocument/2006/relationships/worksheet",
$NewWorkBookRelId)
```

Everything else is optional

Worksheet content

If you put data into a Microsoft Excel worksheet, Excel will automatically convert some data into the format that Excel thinks is best.
For example, Excel will remove leading Zeros of Numbers, change Date/Time Formats or uses the scientific number format for large Numbers and others.
This can go unnoticed in large data sets.

To prevent Excel from converting the data, you must tell Excel to import/store the data in Text format.

There are two ways to store data with Type of Text in an Excel XLSX worksheet package part!

1. Inline strings which are stored inside the XML worksheet package part (file)
    - Provided for ease of translation/conversion
    - Useful in XSLT scenarios
    - Excel and other consumers may convert to shared strings
    - to export the data programmatically into the worksheet

2. Using a shared-strings XML package part as a table with unique strings
    - All worksheets points/links to the strings stored in the shared-strings package part
    - Each unique string is stored once (reduced file size, improved performance)
    - Cells store the 0-based index of the string

Both approaches may be mixed/combined

I will use the inline string approach here in my PowerShell solution, because it is easier to create and maintain.

Example of an Excel XML worksheet part which contains only inline content, formatted as Type of text:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<worksheet xmlns="http://schemas.openxmlformats.org/spreadsheetml/2006/main"
xmlns:r="http://schemas.openxmlformats.org/officeDocument/2006/relationships">
  <sheetData>
    <row>
      <c t="inlineStr">
        <is>
          <t>Name</t>
        </is>
      </c>
    </row>
    <row>
```

```
      <c t="inlineStr">
        <is>
          <t>acrotray</t>
        </is>
      </c>
    </row>
    <row>
      <c t="inlineStr">
        <is>
          <t>Name</t>
        </is>
      </c>
    </row>
  <sheetData>
```

A row is represented as <row>-Element.
A cell is represented as <c>-Element. The type of the cell is defined by the "t" attribute here as type of "inlineStr" which means a type of text.
If the cell has a type of "inlineStr" the <c> node must contain a <is> node.
For a simple string (text) without formatting the <is> node contains a <t> node with the value of the string.

Warning:
By default Excel uses and stores strings into the shared-strings XML package part.
Excel transfers the inline strings into the shared-strings part on save actions!
So, after Excel has converted the data into shared strings, the data cannot easily accessed!

The PowerShell code

There are several golden rules for code design.
Two of them are:

A Function should always be concentrated to solve only one task and not being a Swiss army knife.
A Function and a scripts should always return well defined and structured Objects

So I have divided my PowerShell code into several functions.

New-XLSXWorkBook

Function to create a new empty Excel .xlsx workbook (XLSX package) without a worksheet

Add-XLSXWorkSheet

Function to append a new empty Excel worksheet to an existing Excel .xlsx workbook

Export-WorkSheet

Function to fill an empty existing Excel worksheet with string typed data

These functions are only used internally. So best is to hide these functions.
To hide functions you have these options in PowerShell

- nest functions inside other function (in case of advance functions put it inside the begin block)
- nest functions inside the begin block of an advanced script
- Create a module and specify the public module members with the Cmdlet Export-ModuleMember

I don’t want to force the user of my script to import it as a module.
In the fact that a PowerShell script can look and behave like a function I have decided to nest the functions inside the begin block of the script.
So you can use this script by simply calling it (by its path) and by use of the parameters.

Office Open XML Format Links:

ISO and IEC standards
http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html

Ecma standard 376
http://www.ecma-international.org/publications/standards/Ecma-376.htm

Office Open XML Learn resources:

Exploring the Office Open XML Formats
http://office.microsoft.com/en-us/training/office-open-xml-i-exploring-the-office-open-xml-formats-RZ010243529.aspx?section=1

Editing Documents in the XML
http://office.microsoft.com/en-us/training/open-xml-ii-editing-documents-in-the-xml-RZ010357030.aspx?CTT=1

Good Open XML XLSX Link:

Read and write Open XML files (MS Office 2007)
http://www.developerfusion.com/article/6170/read-and-write-open-xml-files-ms-office-2007/

SpreadsheetML or XLSX
http://officeopenxml.com/anatomyofOOXML-xlsx.php

Tipp:
WindowsBase.dll can even be used in PowerShell 2.0 and 3.0 to create ZIP Files:

PowerShell-ZIP
http://thewalkingdev.blogspot.de/2012/07/powershellzip.html

CodeProject; Use and create Zip archives without external libraries
http://www.codeproject.com/Articles/209731/Csharp-use-Zip-archives-without-external-libraries

PowerShell and the Excel COM Object Model:

For documentation oft he Excel object model search the Microsoft Developer Network (MSDN) for:
" Excel Object Model Reference"
Excel 2003 and 2007:http://msdn.microsoft.com/en-us/library/bb149081%28v=office.12%29.aspx
Excel 2003 and 2007:http://msdn.microsoft.com/en-Us/library/wss56bz7%28v=vs.90%29.aspx
Excel 2013: http://msdn.microsoft.com/en-us/library/office/ff194068.aspx

Article series: Integrating Microsoft Excel with PowerShell by Jeffery Hicks:
http://www.petri.co.il/export-to-excel-with-powershell.htm
http://www.petri.co.il/export-to-excel-with-powershell-part-2.htm
http://www.petri.co.il/export-to-excel-with-powershell-part-3.htm

How Can I Use Windows PowerShell to Automate Microsoft Excel? By Ed Wilson:
http://blogs.technet.com/b/heyscriptingguy/archive/2006/09/08/how-can-i-use-windows-powershell-to-automate-microsoft-excel.aspx