How to build OpenConsole

This repository uses [git submodules]() for some of its dependencies. To make sure submodules are restored or updated, be sure to run the following prior to building:

```
git submodule update --init --recursive
```

OpenConsole.sln may be built from within Visual Studio or from the command-line using a set of convenience scripts & tools in the **/tools** directory:

When using Visual Studio, be sure to set up the path for code formatting. To download the required clang-format.exe file, follow one of the building instructions below and run:

```
Import-Module .\tools\OpenConsole.psm1
Set-MsBuildDevEnvironment
Get-Format
```

After, go to Tools > Options > Text Editor > C++ > Formatting and check "Use custom clang-format.exe file" in Visual Studio and choose the clang-format.exe in the repository at /packages/clang-format.win-x86.10.0.0/tools/clang-format.exe by clicking "browse" right under the check box.

Building in PowerShell

```
Import-Module .\tools\OpenConsole.psm1
Set-MsBuildDevEnvironment
Invoke-OpenConsoleBuild
```

There are a few additional exported functions (look at their documentation for further details):

- `Invoke-OpenConsoleBuild` - builds the solution. Can be passed msbuild arguments.
- `Invoke-OpenConsoleTests` - runs the various tests. Will run the unit tests by default.
- `Start-OpenConsole` - starts Openconsole.exe from the output directory. x64 is run by default.
- `Debug-OpenConsole` - starts Openconsole.exe and attaches it to the default debugger. x64 is run by default.
- `Invoke-CodeFormat` - uses clang-format to format all c++ files to match our coding style.

Building in Cmd

```
.\tools\razzle.cmd
bcz
```

There are also scripts for running the tests:

- `runut.cmd` - run the unit tests
- `runft.cmd` - run the feature tests
- `runuia.cmd` - run the UIA tests
- `runformat` - uses clang-format to format all c++ files to match our coding style.

Running & Debugging

To debug the Windows Terminal in VS, right click on `CascadiaPackage` (in the Solution Explorer) and go to properties. In the Debug menu, change "Application process" and "Background task process" to "Native Only".

You should then be able to build & debug the Terminal project by hitting `F5`.

> 👉 *You will not be able to launch the Terminal directly by running the WindowsTerminal.exe. For more details on why, see #926, #4043*

Configuration Types

Openconsole has three configuration types:

- Debug
- Release
- AuditMode

AuditMode is an experimental mode that enables some additional static analysis from CppCoreCheck.

Updating Nuget package references - Globally versioned

Most Nuget package references in this project are centralized in a single configuration so that there is a single canonical version for everything. This canonical version is restored before builds by the build pipeline, environment initialization scripts, or Visual Studio (as appropriate).

The canonical version numbers are defined in dep/nuget/packages.config. That defines what will be downloaded by nuget.exe. Most Nuget packages also have a .props and/or .targets file that must be imported by every project that consumes it. Those import statements are consolidated in:

- src/common.nugetversions.props
- src/common.nugetversions.targets

When a globally managed version changes all three of those files must be changed in unison.

Updating Nuget package references - Locally versioned

Certain Nuget package references in this project, like `Microsoft.UI.Xaml`, must be updated outside of the Visual Studio NuGet package manager. This can be done using the snippet below.

> *Note that to run this snippet, you need to use WSL as the command uses `sed`. To update the version of a given package, use the following snippet*

```
git grep -z -l $PackageName | xargs -0 sed -i -e
's/$OldVersionNumber/$NewVersionNumber/g'
```

where:

- `$PackageName` is the name of the package, e.g. Microsoft.UI.Xaml
- `$OldVersionNumber` is the version number currently used, e.g. 2.4.0-prerelease.200506002
- `$NewVersionNumber` is the version number you want to migrate to, e.g. 2.5.0-prerelease.200812002

Example usage:

`git grep -z -l Microsoft.UI.Xaml | xargs -0 sed -i -e 's/2.4.0-prerelease.200506002/2.5.0-prerelease.200812002/g'`

Using .nupkg files instead of downloaded Nuget packages

If you want to use .nupkg files instead of the downloaded Nuget package, you can do this with the following steps:

1. Open the Nuget.config file and uncomment line 8 ("Static Package Dependencies")
2. Create the folder /dep/packages
3. Put your .nupkg files in /dep/packages
4. If you are using different versions than those already being used, you need to update the references as well. How to do that is explained under "Updating Nuget package references".

Building the Terminal package from the commandline

The Terminal is bundled as an `.msix`, which is produced by the `CascadiaPackage.wapproj` project. To build that project from the commandline, you can run the following (from a window you've already run `tools\razzle.cmd` in):

```
"%msbuild%" "%OPENCON%\OpenConsole.sln" /p:Configuration=%_LAST_BUILD_CONF%
/p:Platform=%ARCH% /p:AppxSymbolPackageEnabled=false /t:Terminal\CascadiaPackage /m
```

This takes quite some time, and only generates an `msix`. It does not install the msix. To deploy the package:

```
# If you haven't already:
Import-Module tools\OpenConsole.psm1;
Set-MsBuildDevEnvironment;

# The Set-MsBuildDevEnvironment call is needed for finding the path to
# makeappx. It also takes a little longer to run. If you're sticking in powershell,
best to do that.

Set-Location -Path
src\cascadia\CascadiaPackage\AppPackages\CascadiaPackage_0.0.1.0_x64_Debug_Test;
if ((Get-AppxPackage -Name 'WindowsTerminalDev*') -ne $null) {
Remove-AppxPackage 'WindowsTerminalDev_0.0.1.0_x64__8wekyb3d8bbwe'
};
New-Item ..\loose -Type Directory -Force;
makeappx unpack /v /o /p .\CascadiaPackage_0.0.1.0_x64_Debug.msix /d ..\Loose\;
Add-AppxPackage -Path ..\loose\AppxManifest.xml -Register -ForceUpdateFromAnyVersion
-ForceApplicationShutdown
```

Or the cmd.exe version:

```
@rem razzle.cmd doesn't set:
@rem set WindowsSdkDir=C:\Program Files (x86)\Windows Kits\10\
@rem vsdevcmd.bat does a lot of logic to find that.
@rem
@rem I'm gonna hard code it below:
```

```
powershell -Command Set-Location -Path
%OPENCON%\src\cascadia\CascadiaPackage\AppPackages\CascadiaPackage_0.0.1.0_x64_Debug_Te
 ((Get-AppxPackage -Name 'WindowsTerminalDev*') -ne $null) { Remove-AppxPackage
'WindowsTerminalDev_0.0.1.0_x64__8wekyb3d8bbwe'};New-Item ..\loose -Type Directory -
Force;C:\'Program Files (x86)'\'Windows Kits'\10\bin\10.0.19041.0\x64\makeappx
unpack /v /o /p .\CascadiaPackage_0.0.1.0_x64_Debug.msix /d ..\Loose\;Add-
AppxPackage -Path ..\loose\AppxManifest.xml -Register -ForceUpdateFromAnyVersion -
ForceApplicationShutdown
```

(yes, the cmd version is just calling powershell to do the powershell version. Too lazy to convert the rest by hand, I'm already copying from `.vscode\tasks.json`)

Building the package from VS generates the loose layout to begin with, and then registers the loose manifest, skipping the msix stop. It's a lot faster than the commandline inner loop here, unfortunately.