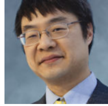
A graphical depiction of the steps in building a C++ executable, with XAML and packaging

devblogs.microsoft.com/oldnewthing/20240531-00

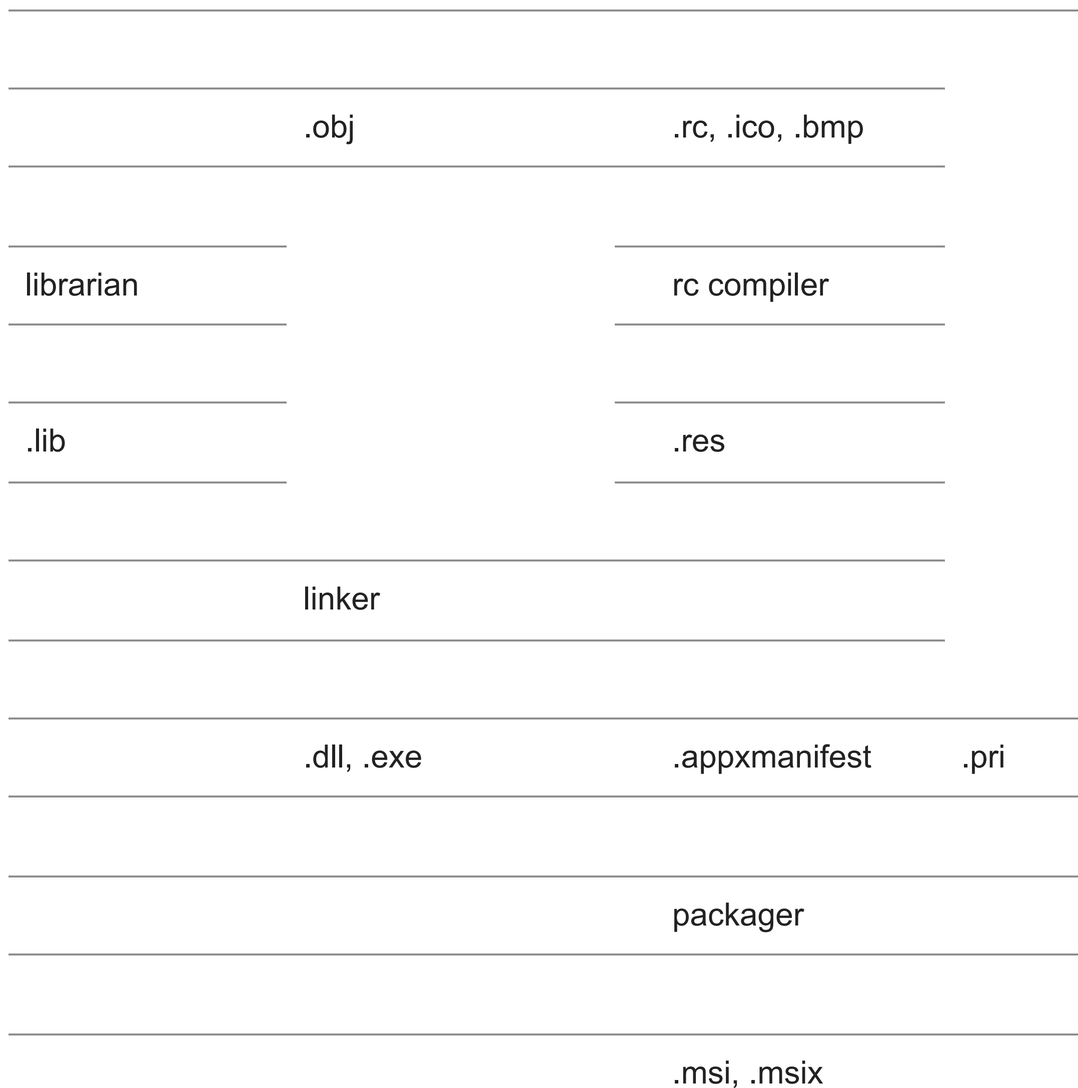
May 31, 2024



Raymond Chen

Last time, we generated a diagram showing how a C++ executable is built, augmented for classic Win32. Let’s add in more stuff: XAML and packaging.

.idl			
MIDL compiler			
	.winmd	.xaml	
	C++/WinRT compiler	XAML compiler	
.h, .cpp		.xbf	.resw
	C++ compiler		makepri



We still see the core of the compiler and linker on the left hand side, happily consuming C++ source files (.h, .cpp) producing a module (.dll, .exe). We just added still more steps that either produce C++ source files from other sources, add additional content to the resulting module, or which package the resulting module.

If there is an .idl file, it is processed by the MIDL compiler, which (depending on what kind of IDL file you provided) may produce a .winmd file, or C source files. If you're processing .idl files for C++/WinRT, then the thing your project uses is the .winmd file. If you're processing .idl files for C++/WRL, then the thing your project uses is the .h file. And if you want your interfaces to be marshallable, you will want the .c file that compiles the proxy stubs.

If you have a .winmd file and are using C++/WinRT, then the C++/WinRT compiler studies the .winmd file and produces .h and .cpp files for your project to use. (C++/WinRT-based projects use these header files instead of the MIDL ones.)

If you have .xaml files, then XAML compiler generates C++ source code for those .xaml files, using the .winmd files to know how to generate binding code. It also generates .xbf files (which I correctly guessed stands for XAML binary format) which are used at runtime to generate the elements you specified in your XAML markup. The C++ source files get compiled into your project, too.

Now that we have generated all the C++ source files, we can go through the usual procss of compiling the C++ source files to object files, and possibly generate a .lib.

If your code uses classic Win32 resources, the resource compiler takes the .rc file and any supporting files (like .ico and .bmp) and produces a .res file, as before.

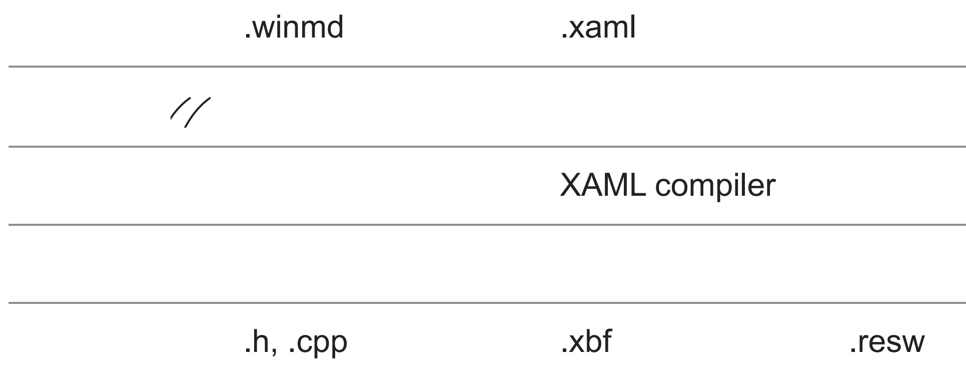
After all the .obj, .lib, and .res files have been produced, we feed them all to the linker, which produces the resulting module, a .dll or .exe.

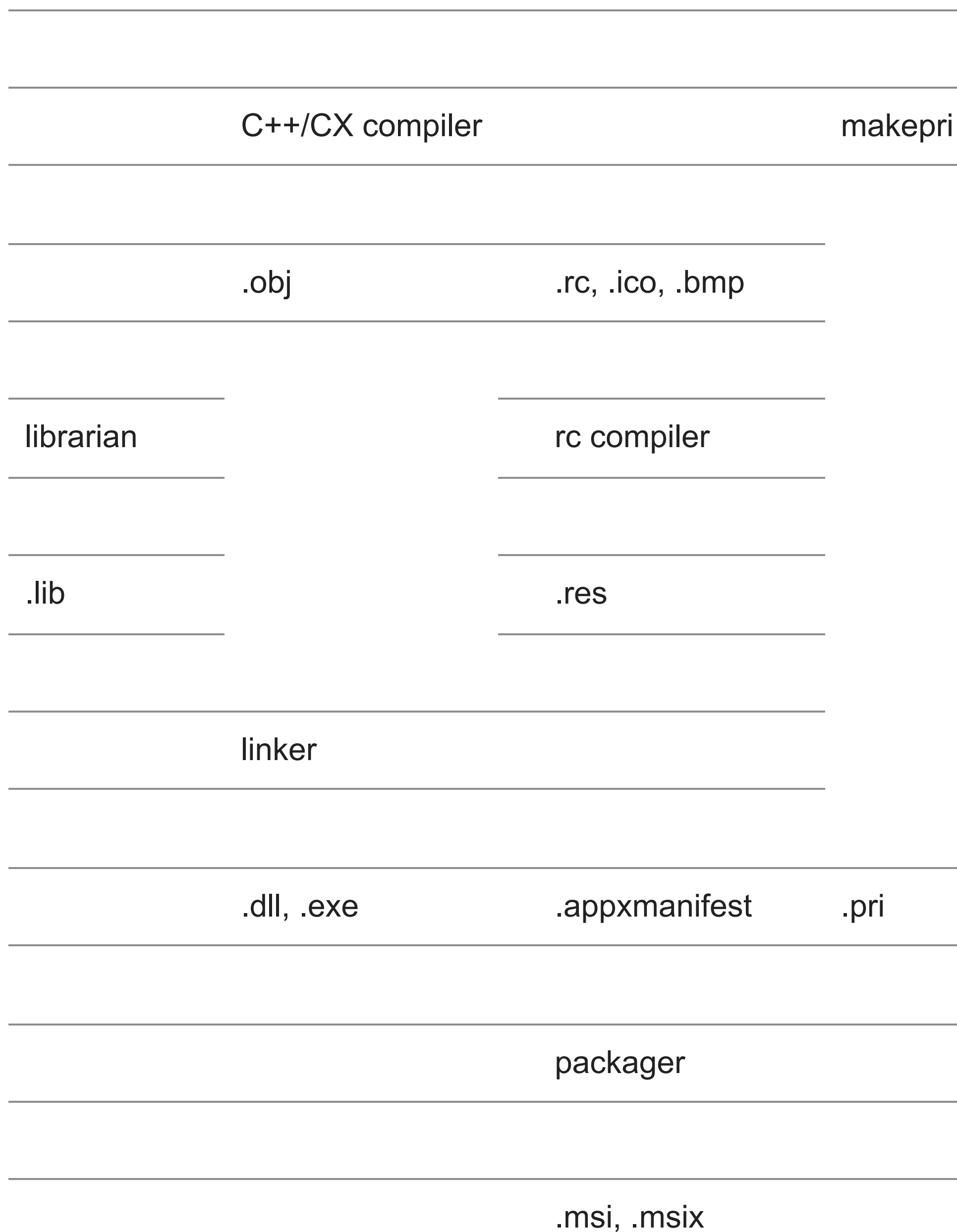
If applicable to your project, the `makepri` program takes a string resource description file .resw, as well as all of the .xbf files, and produces a .pri file, which is yet another binary format for resources.

If your final product requires packaging, then all of the .dll files, .exe files, .pri files, and any other collateral are gathered together, along with your package’s .appxmanifest, and out comes an .msi file or .msix file.

This is a very quick overview of the build flow, but now that you see how the pieces fit together, you will be better-equipped to diagnose build problems. For example, if you get a XAML compiler error, then the things to investigate are the .xaml and .winmd files in your project, since those are the pieces that the XAML compiler uses. There’s no point investigating the manifest file when trying to diagnose a XAML compiler error.

Bonus chatter: If your project is based on C++/CX, then the build flow is different, since C++/CX doesn’t use .idl files but rather generates the .winmd file from your C++ sources. The resulting diagram looks weird because there is a circular dependency!





Instead of generating the .winmd from an .idl file, the .winmd is generated from the C++ sources, and the C++/CX compiler also consumes the .winmd files when compiling the C++ sources. My understanding is that the C++/CX compiler first makes a “metadata pass” where it parses the code only to identify the `ref` classes and their members. It then generates a .winmd from that information, and then later, in the “real compilation” pass, all of the .winmd files are used to provide type information to the C++/CX compiler a second time.

In addition to introducing a weird circular dependency, this style of automatic .winmd generation limits you to the types of things that the automatic .winmd generator understands. The automatic .winmd generator doesn’t understand Windows Runtime versioning or contracts, for example, so you can’t use it to produce a v2 API that is backward compatible

with the v1 API. Once you make a change, all of your clients have to switch to the v2 API; the v1 API doesn't work any more. This is fine for monolithic programs, but it's bad news if you're trying to produce reusable components.