**.PDB File Overview**

The .pdb (or Program DataBase) files generated by VS are just as important as source code to perform effective debugging. They contain all the pertinent debug info that the compiled exe/dll does not; it can be near impossible to debug a build without its matching PDB file.

What info might a PDB file contain? (I say “might” because it can differ depending on the language and compiler options):

* Function names and addresses
* Global/static variable names and addresses
* Parameter and local variable names and offsets of where to find them on the stack
* Type information for variables and data structures
* Frame Pointer Optimization (FPO) data
* Source file names and lines

Differences for .NET PDBs: they only contain source file names and lines, and local variable names. All other data is already contained in .NET metadata, so there’s no need to duplicate it.

If you plan on releasing builds to other people and want to be able to debug those builds, it’s important to set up a Symbol Server to store the associated PDBs. Both VS and WinDBG can access symbol servers. If you also want the source files to automatically sync to the correct version for a build you’re debugging, you can run Source Server Indexing on the PDB file, which embeds the version control commands to pull the correct source versions.

*Creation and Resolution*

PDBs are created by the linker, which also embeds a newly-generated GUID into the PDB file. That same GUID, along with the absolute path of the created PDB file, are both embedded into the compiled exe/dll. This is how the debugger can identify which PDB matches what is being debugged.

When trying to find a matching PDB for debugging, the debugger looks in the following places (in order):

1. The absolute path of the PDB embedded in the exe/dll
2. Same folder as the exe/dll
3. Any local symbol cache folders
4. Any local network symbol servers
5. Any internet symbol servers (e.g. Microsoft)

TIP: To see what PDB version an exe/dll expects, open a VS Native Tools command prompt (or our own IG prompt) and run "dumpbin /headers <path\_to\_exe/dll>" and look for the Debug Directories output.

Example output:

Debug Directories

Time Type Size RVA Pointer

-------- ------- -------- -------- --------

58863B2E cv 58 011D7A94 11D6694 Format: RSDS, {**1DBC8545-F71D-49BF-98BD-DD18B652AEE0**}, 3, **X:\core\phalanx\devel\code\Output\Built\Release\i20\_Windows.pdb**

58863B2E feat 14 011D7AEC 11D66EC Counts: Pre-VC++ 11.00=0, C/C++=2628, /GS=191, /sdl=0, guardN=42

58863B2E coffgrp 39C 011D7B00 11D6700

As a consequence of the linker generating a new GUID every time it runs, even if you make no functional changes to your code and recompile (e.g. you add whitespace somewhere), the previous exe/dll would not be compatible with the new PDB\*. This is why it's important to save the original PDBs from a particular public build if you hope to be able to debug it later - you can't just sync the code back to the proper point, recompile, and hope to use those PDBs for debugging the existing exe/dll.

\*You could probably hex-edit the GUID in either file to force a match, but because compiler/linker determinism isn't guaranteed\*\*, i.e. the resulting code layout may not be the same from build to build, that hack is generally a bad idea.

\*\*Apparently Microsoft has recently added a "deterministic" compiler option to the open-source Roslyn compiler for C# and VB (VS 2015 Update 2)

<https://blogs.msdn.microsoft.com/dotnet/2016/04/02/whats-new-for-c-and-vb-in-visual-studio/>