The PDB DBI (Debug Info) Stream

Introduction

The PDB DBI Stream (Index 3) is one of the largest and most important streams in a PDB file. It contains information about how the program was compiled, (e.g. compilation flags, etc), the compilands (e.g. object files) that were used to link together the program, the source files which were used to build the program, as well as references to other streams that contain more detailed information about each compiland, such as the CodeView symbol records contained within each compiland and the source and line information for functions and other symbols within each compiland.

Stream Header

At offset 0 of the DBI Stream is a header with the following layout:

```
struct DbiStreamHeader {
  int32_t VersionSignature;
  uint32_t VersionHeader;
  uint32_t Age;
  uint16_t GlobalStreamIndex;
  uint16_t BuildNumber;
  uint16_t PublicStreamIndex;
  uint16_t PdbDllVersion;
  uint16_t SymRecordStream;
  uint16_t PdbDllRbld;
  int32_t ModInfoSize;
  int32_t SectionContributionSize;
  int32_t SectionMapSize;
  int32_t SourceInfoSize;
  int32_t TypeServerMapSize;
  uint32_t MFCTypeServerIndex;
  int32_t OptionalDbgHeaderSize;
  int32_t ECSubstreamSize;
  uint16_t Flags;
  uint16_t Machine;
  uint32_t Padding;
};
```

- **VersionSignature** – Unknown meaning. Appears to always be `-1`.
- **VersionHeader** – A value from the following enum.

```
enum class DbiStreamVersion : uint32_t {
  VC41 = 930803,
  V50 = 19960307,
  V60 = 19970606,
  V70 = 19990903,
  V110 = 20091201
};
```

Similar to the PDB Stream, this value always appears to be `V70`, and it is not clear what the other values are for.

- **Age** – The number of times the PDB has been written. Equal to the same field from the PDB Stream header.
- **GlobalStreamIndex** – The index of the Global Symbol Stream, which contains CodeView symbol records for all global symbols. Actual records are stored in the symbol record stream, and are referenced from this stream.
- **BuildNumber** – A bitfield containing values representing the major and minor version number of the toolchain (e.g. 12.0 for MSVC 2013) used to build the program, with the following layout:

```
uint16_t MinorVersion : 8;
uint16_t MajorVersion : 7;
uint16_t NewVersionFormat : 1;
```

For the purposes of LLVM, we assume `NewVersionFormat` to be always `true`. If it is `false`, the layout above does not apply and the reader should consult the Microsoft Source Code for further guidance.

- **PublicStreamIndex** – The index of the Public Symbol Stream, which contains CodeView symbol records for all public symbols. Actual records are stored in the symbol record stream, and are referenced from this stream.
- **PdbDllVersion** – The version number of `mspdbXXXX.dll` used to produce this PDB. Note this obviously does not apply for LLVM as LLVM does not use `mspdb.dll`.
- **SymRecordStream** – The stream containing all CodeView symbol records used by the program. This is used for deduplication, so that many different compilands can refer to the same symbols without having to include the full record content inside of each module stream.
- **PdbDllRbld** – Unknown
- **MFCTypeServerIndex** – The index of the MFC type server in the Type Server Map Substream.
- **Flags** – A bitfield with the following layout, containing various information about how the program was built:

```
uint16_t WasIncrementallyLinked : 1;
uint16_t ArePrivateSymbolsStripped : 1;
uint16_t HasConflictingTypes : 1;
uint16_t Reserved : 13;
```

The only one of these that is not self-explanatory is `HasConflictingTypes`. Although undocumented, `link.exe` contains a hidden flag `/DEBUG:CTYPES`. If it is passed to `link.exe`, this field will be set. Otherwise it will not be set. It is unclear what this flag does, although it seems to have subtle implications on the algorithm used to look up type records.

- **Machine** – A value from the CV_CPU_TYPE_e enumeration. Common values are `0x8664` (x86-64) and `0x14C` (x86).

Immediately after the fixed-size DBI Stream header are `7` variable-length *substreams*. The following `7` fields of the DBI Stream header specify the number of bytes of the corresponding substream. Each substream's contents will be described in detail below. The length of the entire DBI Stream should equal `64` (the length of the header above) plus the value of each of the following `7` fields.

- **ModInfoSize** – The length of the Module Info Substream.
- **SectionContributionSize** – The length of the Section Contribution Substream.
- **SectionMapSize** – The length of the Section Map Substream.
- **SourceInfoSize** – The length of the File Info Substream.
- **TypeServerMapSize** – The length of the Type Server Map Substream.
- **OptionalDbgHeaderSize** – The length of the Optional Debug Header Stream.

- **ECSubstreamSize** – The length of the EC Substream.

Substreams

Module Info Substream

Begins at offset `0` immediately after the header. The module info substream is an array of variable-length records, each one describing a single module (e.g. object file) linked into the program. Each record in the array has the format:

```
struct ModInfo {
  uint32_t Unused1;
  struct SectionContribEntry {
    uint16_t Section;
    char Padding1[2];
    int32_t Offset;
    int32_t Size;
    uint32_t Characteristics;
    uint16_t ModuleIndex;
    char Padding2[2];
    uint32_t DataCrc;
    uint32_t RelocCrc;
  } SectionContr;
  uint16_t Flags;
  uint16_t ModuleSymStream;
  uint32_t SymByteSize;
  uint32_t C11ByteSize;
  uint32_t C13ByteSize;
  uint16_t SourceFileCount;
  char Padding[2];
  uint32_t Unused2;
  uint32_t SourceFileNameIndex;
  uint32_t PdbFilePathNameIndex;
  char ModuleName[];
  char ObjFileName[];
};
```

- **SectionContr** – Describes the properties of the section in the final binary which contain the code and data from this module.

  `SectionContr.Characteristics` corresponds to the `Characteristics` field of the IMAGE_SECTION_HEADER structure.

- **Flags** – A bitfield with the following format:

```
// ``true`` if this ModInfo has been written since reading the PDB.  This is
// likely used to support incremental linking, so that the linker can decide
// if it needs to commit changes to disk.
uint16_t Dirty : 1;
// ``true`` if EC information is present for this module. EC is presumed to
// stand for "Edit & Continue", which LLVM does not support.  So this flag
// will always be be false.
uint16_t EC : 1;
uint16_t Unused : 6;
// Type Server Index for this module.  This is assumed to be related to /Zi,
// but as LLVM treats /Zi as /Z7, this field will always be invalid for LLVM
// generated PDBs.
uint16_t TSM : 8;
```

- **ModuleSymStream** – The index of the stream that contains symbol information for this module. This includes CodeView symbol information as well as source and line information. If this field is -1, then no additional debug info will be present for this module (for example, this is what happens when you strip private symbols from a PDB).
- **SymByteSize** – The number of bytes of data from the stream identified by `ModuleSymStream` that represent CodeView symbol records.
- **C11ByteSize** – The number of bytes of data from the stream identified by `ModuleSymStream` that

represent C11-style CodeView line information.
- **C13ByteSize** - The number of bytes of data from the stream identified by `ModuleSymStream` that represent C13-style CodeView line information. At most one of `C11ByteSize` and `C13ByteSize` will be non-zero. Modern PDBs always use C13 instead of C11.
- **SourceFileCount** - The number of source files that contributed to this module during compilation.
- **SourceFileNameIndex** - The offset in the names buffer of the primary translation unit used to build this module. All PDB files observed to date always have this value equal to 0.
- **PdbFilePathNameIndex** - The offset in the names buffer of the PDB file containing this module's symbol information. This has only been observed to be non-zero for the special `* Linker *` module.
- **ModuleName** - The module name. This is usually either a full path to an object file (either directly passed to `link.exe` or from an archive) or a string of the form `Import:<dll name>`.
- **ObjFileName** - The object file name. In the case of an module that is linked directly passed to `link.exe`, this is the same as **ModuleName**. In the case of a module that comes from an archive, this is usually the full path to the archive.

Section Contribution Substream

Begins at offset `0` immediately after the Module Info Substream ends, and consumes `Header->SectionContributionSize` bytes. This substream begins with a single `uint32_t` which will be one of the following values:

```
enum class SectionContrSubstreamVersion : uint32_t {
  Ver60 = 0xeffe0000 + 19970605,
  V2 = 0xeffe0000 + 20140516
};
```

`Ver60` is the only value which has been observed in a PDB so far. Following this is an array of fixed-length structures. If the version is `Ver60`, it is an array of `SectionContribEntry` structures (this is the nested structure from the `ModInfo` type. If the version is `V2`, it is an array of `SectionContribEntry2` structures, defined as follows:

```
struct SectionContribEntry2 {
  SectionContribEntry SC;
  uint32_t ISectCoff;
};
```

The purpose of the second field is not well understood. The name implies that is the index of the COFF section, but this also describes the existing field `SectionContribEntry::Section`.

Section Map Substream

Begins at offset `0` immediately after the Section Contribution Substream ends, and consumes `Header->SectionMapSize` bytes. This substream begins with an `4` byte header followed by an array of fixed-length records. The header and records have the following layout:

```
struct SectionMapHeader {
  uint16_t Count;    // Number of segment descriptors
  uint16_t LogCount; // Number of logical segment descriptors
};

struct SectionMapEntry {
  uint16_t Flags;         // See the SectionMapEntryFlags enum below.
  uint16_t Ovl;           // Logical overlay number
  uint16_t Group;         // Group index into descriptor array.
  uint16_t Frame;
  uint16_t SectionName;   // Byte index of segment / group name in string table, or 0xFFFF.
  uint16_t ClassName;     // Byte index of class in string table, or 0xFFFF.
  uint32_t Offset;        // Byte offset of the logical segment within physical segment.  If gr
  uint32_t SectionLength; // Byte count of the segment or group.
};
```

```
enum class SectionMapEntryFlags : uint16_t {
  Read = 1 << 0,              // Segment is readable.
  Write = 1 << 1,             // Segment is writable.
  Execute = 1 << 2,           // Segment is executable.
  AddressIs32Bit = 1 << 3,    // Descriptor describes a 32-bit linear address.
  IsSelector = 1 << 8,        // Frame represents a selector.
  IsAbsoluteAddress = 1 << 9, // Frame represents an absolute address.
  IsGroup = 1 << 10           // If set, descriptor represents a group.
};
```

Many of these fields are not well understood, so will not be discussed further.

File Info Substream

Begins at offset `0` immediately after the Section Map Substream ends, and consumes `Header->SourceInfoSize` bytes. This substream defines the mapping from module to the source files that contribute to that module. Since multiple modules can use the same source file (for example, a header file), this substream uses a string table to store each unique file name only once, and then have each module use offsets into the string table rather than embedding the string's value directly. The format of this substream is as follows:

```
struct FileInfoSubstream {
  uint16_t NumModules;
  uint16_t NumSourceFiles;

  uint16_t ModIndices[NumModules];
  uint16_t ModFileCounts[NumModules];
  uint32_t FileNameOffsets[NumSourceFiles];
  char NamesBuffer[][NumSourceFiles];
};
```

NumModules - The number of modules for which source file information is contained within this substream. Should match the corresponding value from the ref:*dbi_header*.

NumSourceFiles: In theory this is supposed to contain the number of source files for which this substream contains information. But that would present a problem in that the width of this field being 16-bits would prevent one from having more than 64K source files in a program. In early versions of the file format, this seems to have been the case. In order to support more than this, this field of the is simply ignored, and computed dynamically by summing up the values of the `ModFileCounts` array (discussed below). In short, this value should be ignored.

ModIndices - This array is present, but does not appear to be useful.

ModFileCountArray - An array of `NumModules` integers, each one containing the number of source files which contribute to the module at the specified index. While each individual module is limited to 64K contributing source files, the union of all modules' source files may be greater than 64K. The real number of source files is thus computed by summing this array. Note that summing this array does not give the number of *unique* source files, only the total number of source file contributions to modules.

FileNameOffsets - An array of **NumSourceFiles** integers (where **NumSourceFiles** here refers to the 32-bit value obtained from summing **ModFileCountArray**), where each integer is an offset into **NamesBuffer** pointing to a null terminated string.

NamesBuffer - An array of null terminated strings containing the actual source file names.

Type Server Map Substream

Begins at offset `0` immediately after the File Info Substream ends, and consumes `Header->TypeServerMapSize` bytes. Neither the purpose nor the layout of this substream is understood, although it is assumed to related somehow to the usage of `/Zi` and `mspdbsrv.exe`. This substream will not be discussed further.

EC Substream

Begins at offset `0` immediately after the Type Server Map Substream ends, and consumes `Header->ECSubstreamSize` bytes. This is presumed to be related to Edit & Continue support in MSVC. LLVM does not support Edit & Continue, so this stream will not be discussed further.

Optional Debug Header Stream

Begins at offset `0` immediately after the EC Substream ends, and consumes `Header->OptionalDbgHeaderSize` bytes. This field is an array of stream indices (e.g. `uint16_t`'s), each of which identifies a stream index in the larger MSF file which contains some additional debug information. Each position of this array has a special meaning, allowing one to determine what kind of debug information is at the referenced stream. `11` indices are currently understood, although it's possible there may be more. The layout of each stream generally corresponds exactly to a particular type of debug data directory from the PE/COFF file. The format of these fields can be found in the Microsoft PE/COFF Specification. If any of these fields is -1, it means the corresponding type of debug info is not present in the PDB.

FPO Data – `DbgStreamArray[0]`. The data in the referenced stream is an array of `FPO_DATA` structures. This contains the relocated contents of any `.debug$F` section from any of the linker inputs.

Exception Data – `DbgStreamArray[1]`. The data in the referenced stream is a debug data directory of type `IMAGE_DEBUG_TYPE_EXCEPTION`.

Fixup Data – `DbgStreamArray[2]`. The data in the referenced stream is a debug data directory of type `IMAGE_DEBUG_TYPE_FIXUP`.

Omap To Src Data – `DbgStreamArray[3]`. The data in the referenced stream is a debug data directory of type `IMAGE_DEBUG_TYPE_OMAP_TO_SRC`. This is used for mapping addresses between instrumented and uninstrumented code.

Omap From Src Data – `DbgStreamArray[4]`. The data in the referenced stream is a debug data directory of type `IMAGE_DEBUG_TYPE_OMAP_FROM_SRC`. This is used for mapping addresses between instrumented and uninstrumented code.

Section Header Data – `DbgStreamArray[5]`. A dump of all section headers from the original executable.

Token / RID Map – `DbgStreamArray[6]`. The layout of this stream is not understood, but it is assumed to be a mapping from `CLR Token` to `CLR Record ID`. Refer to ECMA 335 for more information.

Xdata – `DbgStreamArray[7]`. A copy of the `.xdata` section from the executable.

Pdata – `DbgStreamArray[8]`. This is assumed to be a copy of the `.pdata` section from the executable, but that would make it identical to `DbgStreamArray[1]`. The difference between these two indices is not well understood.

New FPO Data – `DbgStreamArray[9]`. The data in the referenced stream is a debug data directory of type `IMAGE_DEBUG_TYPE_FPO`. Note that this is different from `DbgStreamArray[0]` in that `.debug$F` sections are only emitted by MASM. Thus, it is possible for both to appear in the same PDB if both MASM object files and cl object files are linked into the same program.

Original Section Header Data – `DbgStreamArray[10]`. Similar to `DbgStreamArray[5]`, but contains the section headers before any binary translation has been performed. This can be used in conjunction with `DebugStreamArray[3]` and `DbgStreamArray[4]` to map instrumented and uninstrumented addresses.