The PDB TPI and IPI Streams

Introduction

The PDB TPI Stream (Index 2) and IPI Stream (Index 4) contain information about all types used in the program. It is organized as a header followed by a list of CodeView Type Records. Types are referenced from various streams and records throughout the PDB by their type index. In general, the sequence of type records following the header forms a topologically sorted DAG (directed acyclic graph), which means that a type record B can only refer to the type A if `A.TypeIndex < B.TypeIndex`. While there are rare cases where this property will not hold (particularly when dealing with object files compiled with MASM), an implementation should try very hard to make this property hold, as it means the entire type graph can be constructed in a single pass.

Important

Type records form a topologically sorted DAG (directed acyclic graph).

TPI vs IPI Stream

Recent versions of the PDB format (aka all versions covered by this document) have 2 streams with identical layout, henceforth referred to as the TPI stream and IPI stream. Subsequent contents of this document describing the on-disk format apply equally whether it is for the TPI Stream or the IPI Stream. The only difference between the two is in *which* CodeView records are allowed to appear in each one, summarized by the following table:

TPI Stream	IPI Stream
LF_POINTER	LF_FUNC_ID
LF_MODIFIER	LF_MFUNC_ID
LF_PROCEDURE	LF_BUILDINFO
LF_MFUNCTION	LF_SUBSTR_LIST
LF_LABEL	LF_STRING_ID
LF_ARGLIST	LF_UDT_SRC_LINE
LF_FIELDLIST	LF_UDT_MOD_SRC_LINE
LF_ARRAY	
LF_CLASS	
LF_STRUCTURE	
LF_INTERFACE	
LF_UNION	
LF_ENUM	
LF_TYPESERVER2	
LF_VFTABLE	
LF_VTSHAPE	
LF_BITFIELD	

TPI Stream	IPI Stream
LF_METHODLIST	
LF_PRECOMP	
LF_ENDPRECOMP	

The usage of these records is described in more detail in CodeView Type Records.

Type Indices

A type index is a 32-bit integer that uniquely identifies a type inside of an object file's `.debug$T` section or a PDB file's TPI or IPI stream. The value of the type index for the first type record from the TPI stream is given by the `TypeIndexBegin` member of the TPI Stream Header although in practice this value is always equal to 0x1000 (4096).

Any type index with a high bit set is considered to come from the IPI stream, although this appears to be more of a hack, and LLVM does not generate type indices of this nature. They can, however, be observed in Microsoft PDBs occasionally, so one should be prepared to handle them. Note that having the high bit set is not a necessary condition to determine whether a type index comes from the IPI stream, it is only sufficient.

Once the high bit is cleared, any type index >= `TypeIndexBegin` is presumed to come from the appropriate stream, and any type index less than this is a bitmask which can be decomposed as follows:

```
.---------------------------.------.----------.
|           Unused          | Mode |   Kind   |
'---------------------------'------'----------'
|+32                        |+12   |+8        |+0
```

- **Kind** - A value from the following enum:

```
enum class SimpleTypeKind : uint32_t {
  None = 0x0000,          // uncharacterized type (no type)
  Void = 0x0003,          // void
  NotTranslated = 0x0007, // type not translated by cvpack
  HResult = 0x0008,       // OLE/COM HRESULT

  SignedCharacter = 0x0010,   // 8 bit signed
  UnsignedCharacter = 0x0020, // 8 bit unsigned
  NarrowCharacter = 0x0070,   // really a char
  WideCharacter = 0x0071,     // wide char
  Character16 = 0x007a,       // char16_t
  Character32 = 0x007b,       // char32_t
  Character8 = 0x007c,        // char8_t

  SByte = 0x0068,       // 8 bit signed int
  Byte = 0x0069,        // 8 bit unsigned int
  Int16Short = 0x0011,  // 16 bit signed
  UInt16Short = 0x0021, // 16 bit unsigned
  Int16 = 0x0072,       // 16 bit signed int
  UInt16 = 0x0073,      // 16 bit unsigned int
  Int32Long = 0x0012,   // 32 bit signed
  UInt32Long = 0x0022,  // 32 bit unsigned
  Int32 = 0x0074,       // 32 bit signed int
  UInt32 = 0x0075,      // 32 bit unsigned int
  Int64Quad = 0x0013,   // 64 bit signed
  UInt64Quad = 0x0023,  // 64 bit unsigned
  Int64 = 0x0076,       // 64 bit signed int
  UInt64 = 0x0077,      // 64 bit unsigned int
  Int128Oct = 0x0014,   // 128 bit signed int
  UInt128Oct = 0x0024,  // 128 bit unsigned int
  Int128 = 0x0078,      // 128 bit signed int
  UInt128 = 0x0079,     // 128 bit unsigned int

  Float16 = 0x0046,                 // 16 bit real
  Float32 = 0x0040,                 // 32 bit real
  Float32PartialPrecision = 0x0045, // 32 bit PP real
```

```
  Float48 = 0x0044,                // 48 bit real
  Float64 = 0x0041,                // 64 bit real
  Float80 = 0x0042,                // 80 bit real
  Float128 = 0x0043,               // 128 bit real

  Complex16 = 0x0056,                // 16 bit complex
  Complex32 = 0x0050,                // 32 bit complex
  Complex32PartialPrecision = 0x0055, // 32 bit PP complex
  Complex48 = 0x0054,                // 48 bit complex
  Complex64 = 0x0051,                // 64 bit complex
  Complex80 = 0x0052,                // 80 bit complex
  Complex128 = 0x0053,               // 128 bit complex

  Boolean8 = 0x0030,   // 8 bit boolean
  Boolean16 = 0x0031,  // 16 bit boolean
  Boolean32 = 0x0032,  // 32 bit boolean
  Boolean64 = 0x0033,  // 64 bit boolean
  Boolean128 = 0x0034, // 128 bit boolean
};
```

- **Mode** - A value from the following enum:

```
enum class SimpleTypeMode : uint32_t {
  Direct = 0,         // Not a pointer
  NearPointer = 1,    // Near pointer
  FarPointer = 2,     // Far pointer
  HugePointer = 3,    // Huge pointer
  NearPointer32 = 4, // 32 bit near pointer
  FarPointer32 = 5,  // 32 bit far pointer
  NearPointer64 = 6, // 64 bit near pointer
  NearPointer128 = 7 // 128 bit near pointer
};
```

Note that for pointers, the bitness is represented in the mode. So a `void*` would have a type index with `Mode=NearPointer32, Kind=Void` if built for 32-bits but a type index with `Mode=NearPointer64, Kind=Void` if built for 64-bits.

By convention, the type index for `std::nullptr_t` is constructed the same way as the type index for `void*`, but using the bitless enumeration value `NearPointer`.

Stream Header

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

```
struct TpiStreamHeader {
  uint32_t Version;
  uint32_t HeaderSize;
  uint32_t TypeIndexBegin;
  uint32_t TypeIndexEnd;
  uint32_t TypeRecordBytes;

  uint16_t HashStreamIndex;
  uint16_t HashAuxStreamIndex;
  uint32_t HashKeySize;
  uint32_t NumHashBuckets;

  int32_t HashValueBufferOffset;
  uint32_t HashValueBufferLength;

  int32_t IndexOffsetBufferOffset;
  uint32_t IndexOffsetBufferLength;

  int32_t HashAdjBufferOffset;
  uint32_t HashAdjBufferLength;
};
```

- **Version** - A value from the following enum.

```
enum class TpiStreamVersion : uint32_t {
  V40 = 19950410,
  V41 = 19951122,
  V50 = 19961031,
  V70 = 19990903,
  V80 = 20040203,
};
```

Similar to the PDB Stream, this value always appears to be `V80`, and no other values have been observed. It is assumed that should another value be observed, the layout described by this document may not be accurate.

- **HeaderSize** - `sizeof(TpiStreamHeader)`
- **TypeIndexBegin** - The numeric value of the type index representing the first type record in the TPI stream. This is usually the value 0x1000 as type indices lower than this are reserved (see Type Indices for a discussion of reserved type indices).
- **TypeIndexEnd** - One greater than the numeric value of the type index representing the last type record in the TPI stream. The total number of type records in the TPI stream can be computed as `TypeIndexEnd - TypeIndexBegin`.
- **TypeRecordBytes** - The number of bytes of type record data following the header.
- **HashStreamIndex** - The index of a stream which contains a list of hashes for every type record. This value may be `-1`, indicating that hash information is not present. In practice a valid stream index is always observed, so any producer implementation should be prepared to emit this stream to ensure compatibility with tools which may expect it to be present.
- **HashAuxStreamIndex** - Presumably the index of a stream which contains a separate hash table, although this has not been observed in practice and it's unclear what it might be used for.
- **HashKeySize** - The size of a hash value (usually 4 bytes).
- **NumHashBuckets** - The number of buckets used to generate the hash values in the aforementioned hash streams.
- **HashValueBufferOffset / HashValueBufferLength** - The offset and size within the TPI Hash Stream of the list of hash values. It should be assumed that there are either 0 hash values, or a number equal to the number of type records in the TPI stream (`TypeIndexEnd - TypeEndBegin`). Thus, if `HashBufferLength` is not equal to `(TypeIndexEnd - TypeEndBegin) * HashKeySize` we can consider the PDB malformed.
- **IndexOffsetBufferOffset / IndexOffsetBufferLength** - The offset and size within the TPI Hash Stream of the Type Index Offsets Buffer. This is a list of pairs of uint32_t's where the first value is a Type Index and the second value is the offset in the type record data of the type with this index. This can be used to do a binary search followed by a linear search to get O(log n) lookup by type index.
- **HashAdjBufferOffset / HashAdjBufferLength** - The offset and size within the TPI hash stream of a serialized hash table whose keys are the hash values in the hash value buffer and whose values are type indices. This appears to be useful in incremental linking scenarios, so that if a type is modified an entry can be created mapping the old hash value to the new type index so that a PDB file consumer can always have the most up to date version of the type without forcing the incremental linker to garbage collect and update references that point to the old version to now point to the new version. The layout of this hash table is described in The PDB Serialized Hash Table Format.

CodeView Type Record List

Following the header, there are `TypeRecordBytes` bytes of data that represent a variable length array of CodeView type records. The number of such records (e.g. the length of the array) can be determined by computing the value `Header.TypeIndexEnd - Header.TypeIndexBegin`.

O(log(n)) access is provided by way of the Type Index Offsets array (if present) described previously.