



SPECIAL OFFERS Keep up with new releases and promotions. [Sign up to hear from us.](#)

books, eBooks, and digital learning

[Home](#) > [Articles](#) > [Operating Systems, Server](#) > [Microsoft Servers](#)

PDB File Internals

Aug 6, 2001

 [Print](#)  [Share This](#)

Page 1 of 1

In this excerpt from *Undocumented Windows 2000 Secrets: A Programmer's Cookbook*, Windows programming expert Sven Schreiber reveals Windows secrets he has discovered. Here, he discusses the Microsoft PDB file format and structure.

This article is excerpted from [Undocumented Windows Secrets: A Programmer's Cookbook](#) (Addison Wesley, 2001, ISBN: 0201721872).

One remarkable property of the Windows 2000 .dbg files is that they contain just a very tiny, almost negligible CodeView subsection. Example 1 shows the entire CodeView data included in the ntoskrnl.dbg file, generated by the w2k_dump.exe utility in the \src\w2k_dump directory tree of the sample CD. That's all, folks! Just those lousy 32 bytes. All relevant information is moved to a separate file, so the main purpose of this CodeView section is to provide a link to the real data. As Example 1 suggests, the symbol information has to be searched for in the ntoskrnl.pdb file coming with the Windows 2000 symbol setup.

Example Hex Dump of a PDB CodeView Subsection

```
Address | 00 01 02 03-04 05 06 07 : 08 09 0A 0B-0C 0D 0E 0F | 0123456789AE
-----|-----:-----|-----
00006590 | 4E 42 31 30-00 00 00 00 : 20 7D 23 38-54 00 00 00 | NB10.... }#
000065A0 | 6E 74 6F 73-6B 72 6E 6C : 2E 70 64 62-00 00 00 00 | ntoskrnl.pc
```

The Internal Structure of .pdb Files

Whenever you are faced with an unknown data format, the first thing to do is to run some instances of it through a hex dump viewer. The w2k_dump.exe utility does a good job in this respect. Examining the hex dump of a Windows 2000 PDB file like ntoskrnl.pdb or ntfs.pdb reveals some interesting properties:

- The file seems to be divided into blocks of fixed size—typically 0[ts]400 bytes.
- Some blocks consist of long runs of 1-bits, occasionally interrupted by shorter sequences of 0-bits.
- The information in the file is not necessarily contiguous. Sometimes, the data ends abruptly at a block boundary, but continues somewhere else in the file.

InformIT Promotional Mailings & Special Offers

I would like to receive exclusive offers and hear about products from InformIT and its family of brands. I can unsubscribe at any time.

[Privacy Notice](#)

Email Address

[Submit](#)

- Some data blocks appear repeatedly within the file.

It took me some time until I finally realized that these are typical properties of a compound file. A compound file is a small file system packaged into a single file. The "file system" metaphor readily explains some of the above observations:

- A *file system* subdivides a disk into *sectors* of fixed size, and groups the sectors into *files* of variable size. The sectors representing a file can be located anywhere on the disk and don't need to be contiguous—the file/sector assignments are defined in a *file directory*.
- A *compound file* subdivides a raw disk file into *pages* of fixed size, and groups the pages into *streams* of variable size. The pages representing a file can be located anywhere in the raw disk file and don't need to be contiguous—the stream/page assignments are defined in a *stream directory*.

Obviously, almost any assertions about file systems can be mapped to compound files by simply replacing "sector" by "page", and "file" by "stream". The file system metaphor explains why a PDB file is organized in fixed-size blocks. It also explains why the blocks are not necessarily contiguous. What about the pages with the masses of 1-bits? Actually, this type of data is something very common in file systems. To keep track of used and unused sectors on the disk, many file systems maintain an allocation bit array that provides one bit for each sector (or sector cluster). If a sector is unused, its bit is set. Whenever the file system allocates space for a file, it searches for unused sectors by scanning the allocation bits. After adding a sector to a file, its allocation bit is set to zero. The same procedure is applied to the pages and streams of a compound file. The long runs of 1-bits represent unused pages, while the 0-bits are assigned to existing streams.

The only thing that is left now is the observation that some data blocks reoccur within a PDB file. The same thing happens with sectors on a disk. When a file in a file system is rewritten a couple of times, each write operation might use different sectors to store the data. Thus, it can happen that the disk contains free sectors with older duplicates of the file information. This doesn't constitute a problem for the file system. If the sector is marked free in the allocation bit array, it is unimportant what data it contains. As soon as the sector is reclaimed for another file, the data will be overwritten, anyway. Applying the file system metaphor once more to compound files, this means that the observed duplicate pages are usually left over from earlier versions of a stream that has been rewritten to different pages in the compound file. They can be safely ignored—all we have to care for are the pages that are referred to by the stream directory. The remaining unassigned pages should be regarded as garbage.

With the basic paradigm of PDB files being introduced now, we can step to the more interesting task of examining their basic building blocks. Listing 1 shows the layout of the PDB header. The PDB_HEADER starts with a lengthy signature that specifies the PDB version as a text string. The text is terminated with an end-of-file (EOF) character (ASCII code 0[ts]1A) and supplemented with the magic number 0[ts]0000474A, or "JG\0\0" if interpreted as a string. Maybe these are the initials of the designer of the PDB format. The embedded EOF character has the nice effect that an ignorant user can issue a command like `type ntoskrnl.pdb` in a console window without getting any garbage on the screen. The only thing that will be displayed is the message "Microsoft C/C++ program database 2.00\r\n". All Windows 2000 symbol files are shipped as PDB 2.00 files. Apparently, a PDB 1.00 format exists as well, but it seems to be structured quite differently.

Listing 1 The PDB File Header

```
#define PDB_SIGNATURE_200 \  
    "Microsoft C/C++ program database 2.00\r\n\x1AJG\0"  
  
#define PDB_SIGNATURE_TEXT 40  
  
// - - - - -  
  
typedef struct _PDB_SIGNATURE  
{  
    BYTE abSignature [PDB_SIGNATURE_TEXT+4]; // PDB_SIGNATURE_nnn  
}  
PDB_SIGNATURE, *PPDB_SIGNATURE, **PPPPDB_SIGNATURE;  
  
#define PDB_SIGNATURE_ sizeof (PDB_SIGNATURE)  
  
// -----
```

```

#define PDB_STREAM_FREE -1

// - - - - -

typedef struct _PDB_STREAM
{
    DWORD dStreamSize; // in bytes, -1 = free stream
    PWORD pwStreamPages; // array of page numbers
}
PDB_STREAM, *PPDB_STREAM, **PPPDDB_STREAM;

#define PDB_STREAM_ sizeof (PDB_STREAM)

// -----

#define PDB_PAGE_SIZE_1K 0x0400 // bytes per page
#define PDB_PAGE_SIZE_2K 0x0800
#define PDB_PAGE_SIZE_4K 0x1000

#define PDB_PAGE_SHIFT_1K 10 // log2 (PDB_PAGE_SIZE_*)
#define PDB_PAGE_SHIFT_2K 11
#define PDB_PAGE_SHIFT_4K 12

#define PDB_PAGE_COUNT_1K 0xFFFF // page number < PDB_PAGE_COUNT_*
#define PDB_PAGE_COUNT_2K 0xFFFF
#define PDB_PAGE_COUNT_4K 0x7FFF

// - - - - -

typedef struct _PDB_HEADER
{
    PDB_SIGNATURE Signature; // PDB_SIGNATURE_200
    DWORD dPageSize; // 0x0400, 0x0800, 0x1000
    WORD wStartPage; // 0x0009, 0x0005, 0x0002
    WORD wFilePages; // file size / dPageSize
    PDB_STREAM RootStream; // stream directory
    WORD awRootPages []; // pages containing PDB_ROOT
}
PDB_HEADER, *PPDB_HEADER, **PPPDDB_HEADER;

#define PDB_HEADER_ sizeof (PDB_HEADER)

```

Following the signature at offset 0[ts]2C is a DWORD named dPageSize that specifies the size of the compound file pages in bytes. Legal values are 0[ts]0400 (1KB), 0[ts]0800 (2KB), and 0[ts]1000 (4KB). The wFilePages member reflects the total number of pages used by the PDB file image. Multiplying this value by the page size should always exactly match the file size in bytes. wStartPage is a zero-based page number that points to the first data page. The byte offset of this page can be computed by multiplying the page number by the page size. Typical values are 9 for 1KB pages (byte offset 0[ts]2400), 5 for 2KB pages (byte offset 0[ts]2800), or 2 for 4KB pages (byte offset 0[ts]2000). The pages between the PDB_HEADER and the first data page are reserved for the allocation bit array of the compound file, always starting at the beginning of the second page. This means that the PDB file maintains 0[ts]2000 bytes with 0[ts]10000 allocation bits if the page size is 1 or 2KB, and 0[ts]1000 bytes with 0[ts]8000 allocation bits if the page size is 4KB. In turn, this implies that the maximum amount of data a PDB file can manage is 64MB in 1KB page mode, and 128MB in 2KB or 4KB page mode.

The RootStream and awRootPages[] members concluding the PDB_HEADER describe the location of the stream directory within the PDB file. As already noted, the PDB file is conceptually a collection of variable-length streams that carry the actual data. The locations and compositions of the streams are managed in a single stream directory. Funny as it might seem, the stream directory itself is stored in a stream. I have called this very special stream the "root stream". The root stream holding the stream directory can be located anywhere in the PDB file. Its location and size are supplied by the RootStream and awRootPages[] members of the PDB_HEADER. The dStreamSize member of the PDB_STREAM substructure specifies the number of pages occupied by the stream directory, and the entries in the awRootPages[] array point to the pages containing the data.

The stream directory is composed of two sections: A header part in the form of a PDB_ROOT structure, as defined in Listing 2, and a data part consisting of an array of 16-bit page numbers. The wCount member of the PDB_ROOT section specifies the number of streams stored in the PDB compound file. The aStreams[] array contains a PDB_STREAM entry (see Listing 1) for each stream, and the page number slots follow immediately after the last aStreams[] entry.

Listing 2 The PDB Stream Directory

```

#define PDB_STREAM_DIRECTORY 0
#define PDB_STREAM_PDB      1
#define PDB_STREAM_PUBSYM   7

// - - - - -

typedef struct _PDB_ROOT
{
    WORD    wCount;    // < PDB_STREAM_MAX
    WORD    wReserved; // 0
    PDB_STREAM aStreams []; // stream #0 reserved for stream table
}
PDB_ROOT, *PPDB_ROOT, **PPPDB_ROOT;

#define PDB_ROOT_ sizeof (PDB_ROOT)

```

Finding the page number block associated to a given stream is somewhat tricky because the page directory doesn't provide any cues except the stream size. If you are interested in stream #3, you have to compute the number of pages occupied by streams #1 and #2 to get the desired start index within the page number array. Once the stream's page number list is located, reading the stream data is simple. Just walk through the list and multiply each page number by the page size to yield the file offset, and read pages from the computed offsets until the end of the stream is reached. Isn't it funny: On first sight, parsing a PDB file seemed rather tough. Now it turns out that it is actually quite simple—probably much simpler than parsing a .dbg file. The compound-file nature of the PDB format with its clear-cut random access to stream pages reduces the task of reading a stream to a mere concatenation of fixed-sized pages. I'm really amazed at this elegant data access mechanism!

An even greater benefit of the PDB format becomes apparent if it comes to updating an existing PDB file. Inserting data into a file with a sequential structure usually means reshuffling large portions of the contents. The PDB file's random-access structure borrowed from file systems allows addition and deletion of data with minimal effort, just like files can be modified with ease on a file system media. Only the stream directory has to be reshuffled at times when a stream grows or shrinks across a page boundary. This important property facilitates incremental updating of PDB files. As Microsoft puts it in a Knowledge Base article titled "INFO: PDB and DBG Files—What They Are and How They Work":

"The .PDB extension stands for 'program database.' It holds the new format for storing debugging information that was introduced in Visual C++ version 1.0. In the future, the .PDB file will also hold other project state information. One of the most important motivations for the change in format was to allow incremental linking of debug versions of programs, a change first introduced in Visual C++ version 2.0." (*Microsoft Knowledge Base, article Q121366*)

Now that the internal format of PDB files is clear, the next problem is to identify the contents of their streams. After examining various PDB files, I have come to the conclusion that each stream number serves a predefined purpose. For example, the first stream seems to always contain a stream directory, and the second one contains information about the PDB file that can be used to verify that the file matches an associated .dbg file. For example, the latter stream contains `dSignature` and `dAge` members that should have the same values as the corresponding members of an `NB10` CodeView section. The eighth stream is most interesting in the context of this chapter because it hosts the CodeView symbol information we have been searching for. The meaning of the other streams is still unclear to me and constitutes another vast area for future research.

I am not going to include PDB reader sample code here because this would exceed the scope of this article without being rather interesting. You already know this program—it is the `w2k_dump.exe` utility that I have used to create some of the hex dump examples above. This simple console-mode utility provides a `+p` command line option that enables PDB stream decomposition. If the specified file is not a valid PDB file, the program falls back to sequential hex dump mode.