

Paradox database native .NET reader - Upgraded

for DB-MB combinations

v3_jvd 20250130

Introduction

The above-mentioned publication by mr. Petr Bříza on 'www.CodeProject.com' offers excellent opportunities to read Paradox database files with C# coded applications. The class he designed works really great as long as the database you're using doesn't have any information stored in MB files. That part of his code is just not finished. In addition, the internet offers very few sites with (free) solutions to process such MB files correctly. Yes, we have found sites that offer such code for around 1000 dollars to be payed before you discover that it is fake. Those who offered free yet well-functioning paradox readers like the PdxEditor of mr. Knabe, could not be contacted for help, because their email addresses were not active for long.

Borland's Paradox databases are free and fast.

That's why they are still widely used today and why there's still a need for C# code that can handle Paradox DB-MB combinations. So we decided to give it a try ourselves.

Using the program

The following DB-MB table combinations are included in the corresponding demo project:

"gdpprs.db"	text
"gdpplt.db"	text
"album.DB"	text, graphics
"biolife.db"	text, OLE
"cars.db"	text, OLE Picture
"COUNTRIES.db"	text
"films.db"	text, OLE icon, OLE photo

The program can be activated as a normal console application, e.g. 'ParadoxTest.exe gdpprs.db', see the command file 'startcommand.txt.cmd'. To make the result easier to inspect, it is also written to a log file LogFile.txt.

Unfortunately our code cannot handle OLE (BLOb) fieldtypes! It's because these fields are used by customized applications.

Documentation

Borland himself never published his Paradox database format, but there have been a few people who have had the courage and the time to find out by reverse engineering. This could only be done by adding data over and over again and then tracing the internal changes in the Paradox files.

Hats off

We know of two persons who did so:

- mr. Kevin Mitchell: 'Paradox 4.x File Formats, Revision 1, May 11, 1996'.
- mr. Randy Beck: 'The PARADOX File Structure', updated Feb-23-2002.

Roughly speaking, you could say that Petr Bříza's C# code has operationalized Mr. Randy Beck's findings. Beck only described the DB file structure. Mr. Mitchell figured out the format of the corresponding MB files. A careful and accurate application of Mr. Mitchell's findings was sufficient to obtain well-functioning results.

Hats off to them all!

Paradox database native .NET reader - Upgraded

for DB-MB combinations

v3_jvd 20250130

Our findings

One of the problems we faced was caused by the construction of Mr Kevin Mitchell's document. He tended to spread his findings throughout the document. That is why we have compiled his results in an addendum at the end of this document '**Brief summary of Mr. Mitchell's findings**'.

Our understanding was also hampered by the disbelief as to why Borland's Paradox developers had come up with such what we thought was an unnecessarily complicated structure. Unexpected index inversions, applying low- and high-endian randomly, it looked as if they were afraid for their database design being stolen.

One must also realize that at the time when Mr. Mitchell "cracked" Paradox, CPU's operated at 32 bits, and what was then called 'long' consisted of only 4 bytes. Nowadays we would speak of an Int32 number.

Practical restrictions

Since we didn't have the tools to create and edit Paradox tables ourselves, we were limited in our tests to a very small number of DB-MB table combinations that were freely available on the Internet. We have added these to the files that belong to this article.

In the following paragraphs, we will apply what Mr. Mitchell discovered about the structure of DB - MB file combinations, to Mr. Petr Bříza's code.

The structure of Paradox MB files

A Paradox MB file is subdivided in 0x1k (1000h) blocks. Those 1k blocks are not chained and the block length varies. The header block in the MB file is always 0x1k bytes long. Blocks that follow the header have a length that is a multiple of 0x1k and contain the information referenced by the corresponding DB file.

Memo fields are also known as Blob fields and can hold different types of information: plain or formatted text and graphics in most cases. There are several types of Blob fields, see the addendum section 'Brief summary of Mr Mitchell's findings' at the end of this article.

Memo fields in the DB file

The public method GetStringFromMemo(..) in the class file Db.cs of mr. Petr Bříza retrieves a correct 11_byte long MemoBlobArray from a DB file like *gdpprs.db*.

Originally, nothing was done with this 11_byte array, probably because Mr. Bříza did not know what to do with it. However, this 11_byte array is essential for further processing of the corresponding MB file. To enable that further processing we had to make some small modifications to his class file Db.cs to extend the reachability of some variables to the entire application. All those changes are marked with the string "**// jvd**".

Corresponding memo fields in the MB file

According to Mr. Mitchell, there are four types of Memo fields. We only describe the two types we encountered in the files we had at our disposal. These were the so-called **Type2**- and **Type3**-blocks, the former containing single larger (text or graphic) blobs longer than 2048 bytes, the latter for multiple small blobs, used to hold up to 64 blobs smaller than 2k.

Type2- and Type3-blocks can appear anywhere in the MB file. We haven't dealt to much with the question at what length the one or the other is selected.

Paradox database native .NET reader - Upgraded

for DB-MB combinations

v3_jvd 20250130

For this explanation we use the DB-MB table combination *gdpprs.db / gdpprs.mb* which has been copied with permission from the Windows program GensDataPro of the Dutch Genealogical Society. This combination contains freely available textual information about the members of the Dutch Royal Family plus the location of some photos in the application structure, but not of the photos themselves.

For the explanation of how to retrieve the corresponding data from the MB file, we have always selected the first memo field in the DB file, which refers to a Type2 and a Type3 block in the corresponding MB file.

Type2 blocks

The first 11_byte MemoBlobArray which occurs in the DB file *gdpprs.db* and points at a Type2 block is '3C-FF-40-00-00-22-0E-00-00-01-00', // look in *ProcessType2Block(..)*

It tells that at *lOffset1* (*16384d 0x4000*) in the corresponding MB file *gdpprs.mb*, a blob with length *3618d* can be found as follows:

```
long lOffset1 = MemoBlobArray[3] + 256 * MemoBlobArray[2]; // big endian: 0 + 64*256 = 16384d
int iMemoLen = MemoBlobArray[5] + 256 * MemoBlobArray[6]; // little endian: 34 + 14*256= 3618d
```

The byte value at address *0x4000* (*16384d*) in the MB file is 2, indicating the start of a Type2 block. Also the value 'FF' in the second byte in the MemoBlob Array tells you it's a Type2 block.

Note: mr. Mitchell mentions the

- *lOffset1* as 'Size of block divided by 4k'
- *iMemoLen* as 'Length of the blob'.

Type3 blocks

The first 11_byte MemoBlobArray which occurs in the DB file *gdpprs.db* and points at a Type3 block is '3C-3F-10-00-00-00-03-00-00-01-00'.

It tells that at *lOffset1* (*4096d 0x1000*) in the corresponding MB file *gdpprs.mb*, a blob with length *768d* can be found via *PointerArray* nr *63d*, as follows:

```
int iPtrArrayBlokNr = MemoBlobArray[1]; // PtrArrayNr: 63d
long lOffset1 = MemoBlobArray[3] + 256 * MemoBlobArray[2]; // big endian: 0+16*256=4096d
int iMemoLen = MemoBlobArray[5] + 256 * MemoBlobArray[6]; // little endian: 0+3*256=768d
```

The byte value at address *0x1000* (*4096d*) in the MB file is 3, indicating the start of a Type3 block. Type3 blocks use so-called *PointerArrays* (Mitchell: 'Array of Entries') to define the location of text blobs. A Type3 block can contain up to 64 such *PointerArrays*, located in the area between *0x10* to *0x150*, i.e. *320d* bytes long.

Each pointer array consists of 5 bytes (see addendum). The pointer array's are arranged beginning (*more or less*) at the end of the range of *320d* bytes and going down to the beginning at position *0x10*. The numbering of the array's is also going down from 63. Not all 64 positions need to be used.

Why 'more or less'?

According to mr. Mitchell, the offset (from the start of the Type3 block) of an entry indexed by *i* is calculated as follows: *offset = 12 + (5 * i)*. However, this rule is applied with some elasticity, as shown below in the Type3 hex view of *gdpprs.mb*. The first 5-byte pointer array starts at address *0x1147*. That's *0x1147 - 0x1010 = 0x137* (*311d*) bytes away from the beginning of the *320d* space, in which case *i = 61.6* which is not a valid index.

Our conclusion is that the area with the 5 byte pointer arrays may 'float' in the *320d* byte space of a Type3 block.

Paradox database native .NET reader - Upgraded

for DB-MB combinations

v3_jvd 20250130

If PointerArray[63] looks like (hex) **15-30-01-00-10**, then the blob offset is calculated as follows:

Offset in current 1K block = **0x15** * 16d = 21d * 16d = 336d.

The data length = **0x30** * 16d = 768d happens to be identical to the data length found in the 11_byte MemoBlobArray. Be aware that it might not always be the case!

We think the value from the 11_byte MemoBlobArray should prevail.

Hexview examples of Type2 and Type3 Blobfields in the MB file

Type2 hexview (Single Blob Block)

MemoBlobArray '**3C-FF-40-00-00-22-0E-00-00-01-00**' at address 0x587C in DB file, tells us:

```
long lOffset1 = MemoBLObArray[3] + 256 * MemoBLObArray[2]; // big endian - 0x00+256d*0x40=16384d
int iMemoLen = MemoBLObArray[5] + 256 * MemoBLObArray[6]; // little endian- 0x22+256d*0x0E=3618d
```

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Gedecodeerde tekst
00003FF0	30	0D	0A	39	00	00	00	00	00	00	00	00	00	00	00	00	0..9.....
00004000	02	01	00	22	0E	00	00	01	00	3C	6F	70	6D	30	31	3E	...".<opm01>
00004010	5B	5B	4D	61	72	69	61	20	43	68	72	69	73	74	69	6E	[[Maria Christin
00004020	61	2C	20	50	72	69	6E	73	65	73	20	64	65	72	20	4E	a, Prinses der N
00004030	65	64	65	72	6C	61	6E	64	65	6E	2C	20	50	72	69	6E	ederlanden, Prin
00004040	73	65	73	20	76	61	6E	20	4F	72	61	6E	6A	65	2D	4E	ses van Oranj
00004050	61	73	73	61	75	2C	20	50	72	69	6E	73	65	73	20	76	assau, Prinses v
00004060	61	6E	20	4C	69	70	70	65	2D	42	69	65	73	74	65	72	an Lippe-Biester
00004070	66	65	6C	64	2C	20	77	65	72	64	20	67	65	62	6F	72	feld, werd gebor
00004080	65	6E	20	69	6E	20	50	61	6C	65	69	73	20	53	6F	65	en in Paleis Soe
00004090	73	74	64	69	6A	6B	20	6F	70	20	28	31	38	20	66	65	stdijk op (18 fe
000040A0	62	72	75	61	72	69	20	31	39	34	37	29	2E	20	5A	65	bruari 1947). Ze
000040B0	20	69	73	20	64	65	20	6A	6F	6E	67	73	74	65	20	64	is de jongste d
000040C0	6F	63	68	74	65	72	20	76	61	6E	20	77	69	6A	6C	65	ochter van wijle
000040D0	6E	20	4B	6F	6E	69	6E	67	69	6E	20	4A	75	6C	69	61	n Koningin Julia
000040E0	6E	61	20	65	6E	20	77	69	6A	6C	65	6E	20	50	72	69	na en wijlen Pri
000040F0	6E	73	20	42	65	72	6E	68	61	72	64	2E	20	5A	65	20	ns Bernhard. Ze
00004100	6B	72	65	65	67	20	64	65	20	72	6F	65	70	6E	61	61	kreeg de roepnaa
00004110	6D	20	4D	61	72	69	6A	6B	65	2E	0D	0A	50	72	69	6E	m Marijke...Prin
00004120	73	65	73	20	4D	61	72	69	6A	6B	65	20	77	65	72	64	ses Marijke werd
00004130	20	67	65	62	6F	72	65	6E	20	6D	65	74	20	65	65	6E	geboren met een
00004140	20	6F	6F	67	7A	69	65	6B	74	65	2C	20	64	6F	6F	72	oogziekte, door
00004150	64	61	74	20	68	61	61	72	20	6D	6F	65	64	65	72	20	dat haar moeder
00004160	74	69	6A	64	65	6E	73	20	64	65	20	7A	77	61	6E	67	tijdens de zwang
00004170	65	72	73	63	68	61	70	20	62	65	73	6D	65	74	20	77	erschap besmet w
00004180	61	73	20	6D	65	74	20	72	6F	64	65	20	68	6F	6E	64	as met rode hond
00004190	2E	20	44	65	20	77	61	6E	68	6F	6F	70	20	76	61	6E	. De wanhoop van
000041A0	20	4B	6F	6E	69	6E	67	69	6E	20	4A	75	6C	69	61	6E	Koningin Julian

The 0x1k-block of Type2 e.g. starts at 0x4000.

Byte 0x4000 with value 0x02 shows that this block is of Type 2.

Bytes 0x4001 through 2 tell that this block is no longer than 0x1k.

Bytes 0x4003 through 6: 0E 22 00 00 gives the blob length d3618 .

A Type2 blob should always start at position 0x0009 of the current 1k-block. We found abnormalities in file album.mb

Paradox database native .NET reader - Upgraded

for DB-MB combinations

v3_jvd 20250130

Type3 hexview (Suballocated Block)

MemoBlobArray '3C-3F-10-00-00-00-03-00-00-01-00' at addres 0x128C in DB file, tells us:

```
int iPointerArrayNr = MemoBlobArray[1]; // 63d
long lOffset1       = MemoBlobArray[3] + 256d * MemoBlobArray[2]; // big endian – 4096d
int iMemoLen        = MemoBlobArray[5] + 256d * MemoBlobArray[6]; // little endian – 768d
```

gddprs.MB																		
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Gedecodeerde tekst	
00000FE0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000FF0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001000	03	01	00	16	D6	2A	57	B9	8A	00	49	00	00	00	00	00	...0*W+5.I.....	
00001010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00001090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000010A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000010B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000010C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000010D0	00	00	00	00	00	00	00	00	00	FC	04	01	00	09	FB	01ü.....ü.	
000010E0	01	00	04	F8	03	01	00	06	F7	01	01	00	04	EB	0C	01	...0...÷....ë.	
000010F0	00	05	E7	04	01	00	0A	E6	01	01	00	04	E2	04	01	00	..ç....æ....â...	
00001100	08	DF	03	01	00	09	DE	01	01	00	07	DB	03	01	00	0F	.B....Ë....û....	
00001110	DA	01	01	00	04	D7	03	01	00	09	D6	01	01	00	04	80	Û....*....ô....ë	
00001120	56	01	00	03	7D	03	01	00	0E	7C	01	01	00	04	7A	02	V...}....z.	
00001130	01	00	10	79	01	01	00	04	49	30	01	00	09	46	03	01	...y....I0...F..	
00001140	00	0B	45	01	01	00	04	15	30	01	00	10	00	00	00	00	..E....0....	
00001150	3C	6F	70	6D	30	31	3E	5B	5B	44	69	74	20	69	73	20	<opm01>[[Dit is	
00001160	65	65	6E	20	67	65	64	65	65	6C	74	65	20	76	61	6E	een gedeelte van	
00001170	20	64	65	20	73	74	61	6D	62	6F	6F	6D	20	76	61	6F	de stamboom van	

Byte 0x1000 shows that this 1k-block is of Type 3.

Type 3 may hold up to 64 contiguous so-called PointerArrays, each with a length of 5 bytes. They are in the 320-byte area between addresses 0x0010 and 0x0150. The number of pointerArrays can be anything between 1 and 64. The indexes of the PointerArrays count down from 63 to 0 at the end of the 320-byte area.

Strange enough PointerArray[63] itself does not have to end in address 0x014F as is the case in left hexview.

PointerArray[63] has hex values: **15-30-01-00-10** and points at a blob starting at **0x15 * d16 = 21 * 16 = d336** with length of **0x30 * d16= d48 * d16 = d768** (Note: looks superfluous, value is also contained in the MemoBlobArray).

Limitations

We had just a few Paradox DB-MB table combinations available for testing. One combination contained a lot (219) of Type3 memofields, 2 of Type2, 14 of Type0 and 0 of the large blob type of Type4. We did not investigate the Type0 and Type4 blocks.

We also didn't explore how to process:

- PX, X** and Y** files
- OLE fields

Weak points of Paradox

The testfile Album.mb contains Type2 blocks with graphics for different picture types: jpeg, jpg, gif, bmp as can be seen with a hex-viewer.

In method ProcessBlobType(..) under **if (mbPathFile.ToLower().Contains("album") == true)** is shown that the Type2 offset needs to be increased to higher than 9 values to capture some images. It shows how cripple Paradox can be.

Closer inspection with a hex viewer of that same file reveals numerous orphaned byte blobs with no further features. We mentioned already the 'floating' 5 byte pointer arrays of Type3 blocks.

Conclusion

So let's say: we are on the right track with the data extraction out of Paradox DB-MB table combinations. There is probably still a lot to discover and surprises cannot be ruled out because of Borlands not so sleek design of Paradox. Also other user experiences stated that these Paradox databases can easily be corrupted.

Our conclusion: Borlands Paradox is not a KISS design (Keep it Simple and Sound).

Paradox database native .NET reader - Upgraded

for DB-MB combinations

v3_jvd 20250130

Addendum - Brief summary of Mr. Mitchell's findings:

Blob fields in DB File:

- variable length
- uses an extra 10 bytes in the DB record
- when the entire Blob will fit in the leader, the blob is stored in the leader and is not written to the MB file.
- In a DB record, a blob field is stored as a fixed-length data field (called the **leader**) followed by 10 bytes with the following fields:
 - offset (32bits)
 - index value
 - blob length (ulong)
 - modification number from the MB file header (unsigned short integer (16 bits))
- Binary fields have zero-length **leader**
- Memo fields have a leader of at least 1 byte length.
- Numeric data in a **DB** record stored in *modified big endian format*
- The 10 bytes **leader** of blob information are stored in *little endian*
- MB_Offset: first four bytes after/of (?) the leader, locates the blob data in the MB file.
- MB_Offset = 0 then the entire blob is contained in the leader.
- MB_Index: is the low-order byte from MB_Offset - {Change the low-order byte of MB_Offset to zero.???}
- MB_Index = FFh, then MB_Offset contains the offset of a **Type 02 block** in the MB file.
- MB_Index != FFh, then
 - MB_Offset contains the offset of a Type 03 block in the MB file
 - MB_Index contains the index of an entry in the Blob Pointer Array in the Type 03 block.

The MB File:

- header block: 4k (1000h) long
- following blocks have a length that is a multiple of 4k.
- Each block has the following information in the **first three** bytes:
 - UB (Unsigned byte): Record Type
 - 00 - Header block
 - 01 - ??? // jvd: what about Type 01?
 - 02 - Single blob block: allocated for a blob over 2k bytes long.
The length of the data block is the smallest multiple of 4k that is larger than the blob.
 - 03 - Suballocated block: One 4k block may be suballocated (record Type 03) to hold up to 64 small (under 2k) blobs.
 - 04 - Free block
 - US (unsigned short integer): Number of 4k chunks in this block.
The maximum size is FFFFh x 1000h = 65,535 x 4096.
This is 256 megabytes (the maximum length of a blob).

Type3 MemoBlob fields:

- Each PointerArray entry is 5 bytes long and has the following format.

Hex Offset	Field Type	Description
---------------	---------------	-------------

000000	UB	Data offset divided by 16. The offset is measured from start of the 4k block. If this is zero, then the blob was deleted and the space has been reused for another blob (which is associated with another entry in the array).
000001	UB	Data length divided by 16 (rounded up)
000002	US	Modification number from blob header. This is reset to 1 by a table restructure.
000004	UB	Data length modulo 16. If this is zero, then the associated blob has been deleted and the space can be reused. For an active blob, this value will be between 01h and 10h.

- For example, if an array entry looks like: 25-03-0F-00-07 then the data associated with the entry starts at offset 0250h (25h times 10h) and has 10h times 03h bytes allocated (48 bytes).
- The actual data length is 27h (39 bytes) because there are only 7 bytes of data in the last 16 byte chunk.
- The modification number is in little endian format and is 000Fh (15).