

VB6 Record Descriptors: UDT Layout, Destruction and ANSI Translation Tables

David Zimmer (sandsprite.com), with Claude (Anthropic) · October 2026 · companion to *VB6 Variable Initialization and Teardown*

Abstract. Every user-defined type (UDT) in a VB6 program has a compiler-generated *record descriptor*. The runtime uses it to copy, destroy and marshal the record. These descriptors share their binary format with the per-module, per-object and per-procedure teardown tables (RESDESCTBL) described in the companion paper. We give a field-level specification of that format, derived by compiling 114 targeted VB6 programs and diffing the output. The specification is checked in three ways:

- **Field offsets:** against the running programs.
- **The ANSI layout:** by capturing what VB actually passes to a `Declare`.
- **The descriptors:** by an independent encoder that predicts each descriptor from the VB source alone and reproduces all 114 byte-for-byte.
- **The runtime itself:** against the MSVBVM60.DLL code that sizes, constructs and destroys these entries.

1. Introduction

Record opcodes in VB6 P-code, such as `AssignRecord`, `EraseDestruct` and `DestructRecord`, take an operand that indexes the procedure's constant pool. The pool slot holds a pointer to a record descriptor. Without decoding that structure, a decompiler can show *that* a record was copied, but not *what* the record contains. The descriptor is in fact a compact type description: it lists every field that holds a reference, every array with its bounds, every nested record, and (when needed) a second, ANSI layout of the whole record.

The companion paper showed that module and class data are torn down by walking a RESDESCTBL found at `ObjectInfo.aPublicBytes`. Here we show that RESDESCTBL and the record descriptor are the same format. The procedure-local teardown table embedded in every `ProcDscInfo` is a third instance of it.

2. Method

All results come from VB6 SP6 (VB6.EXE 6.00.8176) compiling to P-code from the command line. The loop is the same one used to build the P-code opcode reference:

- **Generate.** A Python harness emits one project per test case. Each project declares a `Type T` with the fields under test, exercises it (copy, `ReDim`, `Erase`, module, `Static` and local instances), and records ground truth at run time: `LenB` plus each field's `VarPtr` offset.

- **Compile and disassemble.** `VB6.EXE /make` builds the binary. `vbdec`, driven over the Running Object Table, disassembles it and reports the pool slot behind every `udtPool` operand, along with each object's `aPublicBytes / aStaticBytes`.
- **Decode.** A small PE reader pulls the descriptor bytes, which are then parsed by the grammar in this paper.
- **Predict independently.** A separate encoder computes, from the VB source text only, the Unicode layout, the ANSI layout and the exact descriptor the compiler should emit. Any byte that differs is a failure. Pointers cannot be predicted, so pointed-to descriptors are checked recursively against the nested type instead.
- **Cross-check against the runtime.** The grammar was then compared with the runtime's own code, using a symbol-bearing IDA database of `MSVBVM60.DLL` (the 1998 build, image base `0x66000000`). `RESDESC::CbResdesc` computes the size of every entry type, and `RESDESCtbl::ConstructInst`, `DestructItem` and `DestructInstExceptSpecial` walk the tables. That code is the authority on what the format can contain. It confirmed every rule derived from the probes, explained two header fields the probes could only describe, and exposed one case the probes had missed (dynamic arrays of fixed-length strings, §6).
- **Confirm at run time.** For records that need ANSI translation, a probe passes the record to `RtlMoveMemory` through a `Declare` and dumps the bytes VB actually marshalled.

The cases cover:

- every intrinsic type and every reference kind (String, Variant, Object, IUnknown, Collection, StdFont, user classes including `As New`);
- fixed-length strings, with fixed arrays of 1–3 dimensions and negative or non-zero lower bounds;
- dynamic arrays;
- nested records three levels deep, with arrays of records at every level;
- class, form, module, `Static` and local scope;
- a public UDT in an ActiveX DLL, and a struct imported from a type library.

Evidence markers used below:

- **[V]** reproduced byte-exact in all lab cases.
- **[R]** also confirmed at run time.
- **[S]** asserted over the scope scenarios.
- **[O]** observed, but with few samples.
- **[RT]** read directly from the runtime code that consumes the field.

3. Where descriptors live

Instance	Found at	Entry offsets relative to
Record descriptor (one per Type)	Const pool, via <code>udtPool:N</code> ; also pointers from other descriptors	start of the record
Module RESDESC_TBL	<code>aPublicBytes</code> (module-level variables), <code>aStaticBytes</code> (procedure Statics)	module data block
Class / Form RESDESC_TBL	<code>aPublicBytes</code> of the object	object instance; member data begins at +0x34 [S]
Procedure frame table	ProcDscInfo + 0x18	EBP, as a signed 16-bit offset [S]

Plain records (no references, no strings) are copied with `CopyBytes`. Their descriptor is still emitted, as a 12-byte header with no entries, but no P-code refers to it.

4. The header

Off	Type	Field	Meaning
+0	u16	cbDesc	Total size of the descriptor in bytes [V]
+2	u16	cbRecord	<code>LenB</code> of the record [V][R]; data size for RESDESC_TBLs; 0 in frame tables
+4	u16	cConstruct	Number of entries the runtime must <i>construct</i> when the procedure is entered (local fixed arrays). 0 in record descriptors and module tables [RT]
+6	u16	cEntries	Number of entries [V]
+8	u16	cbAnsi	Size of the record's ANSI layout [V][R]; 0 in RESDESC_TBLs
+10	u8	align	Maximum member alignment, capped at 4 for VB types [V]; 0 in RESDESC_TBLs
+11	u8	flags	0x01 (frame tables) the first entry is the function's return value · 0x04 always set (record) · 0x08 has members needing destruction · 0x10 has an ANSI translation table · 0x20 contains string data [V]

Type T : p As Long : x As String : z As Long : End Type	LenB = 12
10 00 0C 00 00 00 01 00 0C 00 04 2C cbDesc=16 cbRecord=12 n=1 cbAnsi=12 align=4 flags	
04 00 01 00	+4 String

Figure 1. The smallest non-trivial descriptor. Flags 0x2C = record | needs destruction | contains strings.

5. Entries and type codes

Entries are packed back to back in declaration order. Each one begins with `u16` offset, `u16` type. The low five bits of the type select the base kind; the higher bits are modifiers [V]:

Code	Kind	Following fields	Size
0x01	String (BSTR)	— (or u16 0, u16 ansiOff when 0x2000 is set)	4 / 8
0x02	Variant	as String	4 / 8
0x03	Object reference, of any interface or class	as String	4 / 8
0x05	Array: +0x80 fixed (embedded), +0x100 dynamic	u16 0, u16 ansiOff, element, [SAFEARRAY template, trailer]	var.
0x09	Nested record: +0x20 inner needs destruction, +0x40 inner contains strings	u16 x, u16 ansiOff, u32 pDesc, 16 reserved bytes	28
0x0A	Fixed-length string	u16 0, u16 ansiOff, u16 length	10
0x0B	Plain bytes (translation only)	u16 cb, u16 ansiOff	8
0x04	Fixed-length String <i>local</i> longer than 64 characters. The frame holds a BSTR allocated (zero-filled) at entry and freed at exit; up to 64 characters the string lives inline in the frame and has no entry. The threshold was probed at 63, 64, 65 and 66 characters.	u16 x, u16 ansiOff, u16 len (len at +8, as 0x0A)	10
0x06	Array lock, released with SafeArrayUnlock	—	4
0x08	Accepted by the runtime but never produced by a probe; no cleanup	—	6
+0x200	On an array: constructed at entry (a local fixed array, entry type 0x0005; counted by cConstruct)		
+0x1000	On a record: the instance is held in ANSI form (the Declare temporary) and is destroyed with __vbaRecDestructAnsi		
+0x2000	Entry carries an explicit ANSI offset. In RESDESCtbls that offset is 0xFFFF, meaning not applicable.		
+0x400	Raises the minimum entry size to 6 in CbResdesc; never produced by a probe.		

Object, IUnknown, Collection, StdFont and user classes all encode as 0x03. As New leaves no trace in the descriptor, because auto-instantiation is emitted as code at each access site [v]. The 16 bytes after every pDesc were zero in every sample, including the DLL and type-library cases.

6. Arrays

A **dynamic array** field (a() As X) is a pointer to a SAFEARRAY. Its entry carries only the element kind (u16), or for record elements u16 kind, u16 0, u32 pDesc, 16 reserved. Numeric elements, and plain-record elements, are written as kind 0. The array still gets an entry, because its storage must be freed.

A **fixed array** is stored inline in the record. Its entry embeds a complete SAFEARRAY image, which lets the runtime treat the inline storage as a real SAFEARRAY when walking it [V]:

```
Type T : p As Long : x(3 To 5) As String : z As Long : End Type      LenB = 20

38 00 14 00 00 00 01 00 14 00 04 2C      header: cbDesc=56 n=1 flags=0x2C
04 00 85 00 00 00 00 00      +4 fixed array (0x85), ansiOff 0
01 00 00 00 00 00 00 00      element: String, u32 0
01 00 96 01 04 00 00 00 00 00 00 00 00 00 00 00      SAFEARRAY: cDims=1 fFeatures=0x196 cbElements=4
03 00 00 00 03 00 00 00      rgsabound[0]: cElements=3 lbound=3
08 00 00 00      trailer: VT_BSTR
```

Figure 2. An embedded fixed array with a non-zero lower bound.

Further details of the fixed-array entry:

- **Bounds order.** Multi-dimensional bounds are stored rightmost dimension first, as in OLE. For example, (-2 To 2, 1 To 3) is stored as {3,1},{5,-2}.
- **fFeatures** identify the element type:
 - 0x196 String (BSTR)
 - 0x896 Variant
 - 0x456 Object, Collection and class types (DISPATCH + HAVEIID)
 - 0x256 IUnknown
 - 0x096 plain types and fixed-length strings
 - 0x016 records
- **Trailer.** Non-record arrays end with a u32 trailer: the VARTYPE, the IID pointer for object arrays, or the constant 0x20 for fixed-length-string arrays. The meaning of 0x20 is unknown.
- **Fixed-length-string elements.** The element's u32 holds the string length, and cbElements is twice it.
- **Dynamic arrays of fixed-length strings** (a() As String * n) carry u16 0, u16 n after the element type, giving a 14-byte entry. The runtime's CbResdesc showed this case, which the original probe set did not contain; two added probes confirmed it [RT][V].
- **Numeric fixed arrays** in untranslated records get no entry at all. Their geometry exists only in a standalone copy of the same template, referenced by AryInRecLdRf / AryInRecLdPr when the program indexes the field.

7. Nested records

A nested record entry points to the inner type's own descriptor, so the structure is a tree. The type code says what the runtime must do with the inner record:

- **0x29:** destroy it.
- **0x49:** translate it (it holds fixed-length strings).
- **0x69:** both.

A plain inner record normally gets no entry. It appears only inside a translated region, as `0x2009` with a byte count and a NULL descriptor pointer. Arrays of records reuse the same element form, with `fFeatures=0x16` and `cbElements` equal to the inner record size.

```
Type TInS : s As String : n As Long : End Type
Type T      : p As Long : x As TInS : z As Long : End Type           LenB = 16

30 00 10 00 00 00 02 00 10 00 04 3C          cbDesc=48 n=2 cbAnsi=16 flags=0x3C
04 00 69 20 00 00 04 00 80 14 40 00          +4 record 0x2069, ansi0ff 4, -> 00401480
00 00 00 00 00 00 00 00 00 00 00 00 00 00    (reserved)
0C 00 0B 20 04 00 0C 00                      +12 plain 4 bytes -> ANSI +12

00401480: 10 00 08 00 00 00 01 00 08 00 04 2C 00 00 01 00      TInS: +0 String
```

Figure 3. A nested record holding a String. Because the inner type contains string data, the outer record carries a translation table (§8).

8. ANSI translation tables

When a record is passed to a `Declare`, VB converts it to an ANSI copy and back again (the `CRec2Ansi` / `CRec2Uni` opcodes). Two things differ between the layouts:

- A fixed-length string of n characters occupies n bytes in ANSI and $2n$ in Unicode.
- Strings inside nested records must be converted.

The compiler therefore adds a translation table to the descriptor, using these rules `[V][R]`:

1. **Trigger.** A translation table is created by a fixed-length string, or by a nested record (or array of records) whose type contains string data. Arrays of `String` alone do not create one.
2. **Flagging.** From the first triggering member onward, every member is emitted with flag `0x2000` and its ANSI offset. Plain members each get a separate `0x200B` block; blocks are never merged. Plain fixed arrays become full `0x2085` entries with element kind `0x200B` and their `VARTYPE` as the trailer.
3. **Prefix.** Members before the first trigger keep the unflagged form. They sit at the same offsets in both layouts, so they are covered by copying the prefix.
4. **Size.** `cbAnsi` in the header is the size of the ANSI copy.

```
Type T : x As String * 5 : s As String : z As Long : End Type           LenB = 20, cbAnsi = 16

26 00 14 00 00 00 03 00 10 00 04 3C          fixed string, Unicode +0 -> ANSI +0, 5 chars
00 00 0A 20 00 00 00 00 05 00                String,          Unicode +12 -> ANSI +8
0C 00 01 20 00 00 08 00                      4 plain bytes, Unicode +16 -> ANSI +12
10 00 0B 20 04 00 0C 00

bytes received by RtlMoveMemory (16): 41 42 43 44 20 | 00 44 00 | 84 90 5D 00 | 44 33 22 11
                                     "ABCD "      pad      ANSI BSTR*      z = &H11223
```

Figure 4. The descriptor and the runtime marshalling of the same record agree byte for byte.

A layout detail confirmed at run time is that fixed-length strings are byte-aligned even in the Unicode layout. In `Byte`, `String * 1`, `Byte` the fields sit at offsets 0, 1 and 3, and `LenB` is 4 [R]. In the probes, a translated record that also held an Object member (set to Nothing) caused an access violation when passed through a `Declare` under SP6. That behaviour is recorded here but has not been investigated.

9. RESDESC_TBL: module, object and frame tables

The teardown tables use the same grammar, with these differences [5]:

- `cbAnsi` and `align` are 0.
- `flags` is 0x10 if any entry is translated. In frame tables, bit 0x01 means *the first entry is the function's return value*. It is set for Functions returning a String, Variant, Object, array or record. `ConstructInst` and `DestructInstExceptSpecial` skip that entry, so the return value outlives the frame teardown [RT]. Functions returning plain types and Subs have 0.
- Translated entries carry ANSI offset 0xFFFF.

```
Dim gT As T, gA() As T      (T as in Figure 1)

48 00 18 00 00 00 02 00 00 00 00 10      cbDesc=72 cb=24 n=2 flags=0x10
00 00 69 20 00 00 FF FF 44 12 40 00 + 16  +0 record T -> 00401244
0C 00 05 21 00 00 FF FF 69 20 00 00 44 12 40 00 + 16
                                         +12 dynamic array of record T
```

Figure 5. Module `aPublicBytes`: the paper's RESDESC_TBL is a record descriptor over the module data block.

Procedure frames. `ProcDscInfo` has a fixed 0x18-byte head followed by a RESDESC_TBL describing the destructible locals. The table lists records, dynamic arrays, Strings and Objects, in no particular order; `ByRef` parameters are not included. Its offsets are negative EBP displacements, the same encoding as the `%a` operands. `ProcDscInfo +0x0A` holds `0x18 + cbDesc` (the length of the head plus the table) in every procedure examined. Disassemblers that label +0x18 onward as fixed scalar fields are reading this table's header and first entry.

The procedure prolog (`ProcCallEngine` → `ProcEntry`) calls `ConstructInst` when the table's `cConstruct` word (`RTMI+0x1C`) is non-zero. That is how a local `Dim a(5) As Long` gets its `SAFEARRAY` data before the first statement runs. The exit path (`lblExitProcCommon`) calls `DestructInstExceptSpecial` whenever `cbDesc` is larger than the 12-byte header [RT]. The remaining `ProcDscInfo` fields are documented separately: argument and frame sizes, `Friend` and `Class_Initialize` flags, the `Erl` table, and the long-statement table used by `Resume`.

```
Sub Main: Dim m As T, ma() As T, k As T      (ProcDscInfo+0x18)

64 00 00 00 00 00 03 00 00 00 00 10      cbDesc=100 n=3 flags=0x10
48 FF 69 20 00 00 FF FF F0 11 40 00 + 16  [ebp-0B8h] record T
64 FF 69 20 00 00 FF FF F0 11 40 00 + 16  [ebp-09Ch] record T
60 FF 05 21 00 00 FF FF 69 20 00 00 F0 11 40 00 + 16  [ebp-0A0h] T()
```

Objects. Class and form tables give offsets from the object, with member data starting at +0x34. Their `cbRecord` exceeds the end of the last member by 12 bytes in both samples. A module's data block is likewise 4–8 bytes larger than its variables; this compiler-owned tail is not decoded.

9a. Validation on a production binary

The decoder was run over a real program: the p-code build of PDFStreamDumper, 516 KB, 612 procedures. All 647 tables (612 procedure frames plus every module and object table) decode to exactly their declared size, and the VB and Python implementations agree on every one. That binary used several codes no lab case had produced:

- **Array locks (0x06):** 19 entries.
- **String buffers (0x04):** 3 entries. They match the three `Dim sShortFile As String * 67` locals in its source, which is what led to the 64-character threshold above.
- **Local fixed arrays:** 0x0005 and 0x0285.
- **ANSI-instance records:** 0x3049 and 0x3069.

It also showed two things a decoder must not assume:

- **Fixed arrays:** every non-dynamic array entry carries a `SAFEARRAY` template, whether or not it is flagged 0x80.
- **Fields left uninitialised:** in frame-table entries for `Declare temporaries`, the compiler does not initialise the fields the runtime ignores. A plain record's descriptor slot held 0xBF8, and its 16 reserved bytes held non-zero data. A record's descriptor pointer should only be followed when the type code carries 0x20 or 0x40 [RT].

10. Records from type libraries

A struct imported from a referenced type library gets its own descriptor, with align byte 0. It also keeps its natural alignment: a struct holding a `Double` forces 8-byte alignment on the enclosing VB Type [0]. VB then emits a project-level block `{&LIBID, &RecordGUID, u32 major, u32 minor}`, which holds exactly the arguments to `GetRecordInfoFromGuids`. That block is referenced from just before the first `ObjectInfo`. The descriptor itself never contains the GUID. A Public UDT exported from an ActiveX DLL class produced an ordinary descriptor and no such block.

11. Consuming opcodes

Opcode	Operand	Use
FE F5 AssignRecord	record descriptor	<code>a = b</code> : deep copy driven by the entry list
FE 60 EraseDestruct	record descriptor	Erase of an array of records
FF 2A DestructRecord	record descriptor	explicit record teardown
FF 06/07 AryInRecLdPr/Rf	standalone SAFEARRAY template + dimension count	indexing a fixed array inside a record
1F / 20 CRec2Ansi / CRec2Uni	record descriptor	Declare marshalling via the translation table
FF 03/04 StAryRecMove/Copy	record descriptor	assignment of arrays of records
FE F6 DestructAnsiOFrame	frame temp + record descriptor	frees the ANSI copy made for a Declare call
FF 16–19 Get/PutRecOwn3/4	not a record descriptor: the Put/Get layout (§11a)	Get / Put of a record

11a. Put/Get file layouts

Get #f and Put #f of a record holding strings, Variants or dynamic arrays compile to `GetRecOwn3/4` and `PutRecOwn3/4`. Their pool operand is not a record descriptor. It points to a second structure: a small serialization program that tells the runtime how to walk the record and what to write. An encoder that predicts it from the VB source reproduces all 72 probes byte-for-byte [v]. The bytes each probe wrote to disk confirm the meaning of every operation.

```
header  u32 cbUnpadded    end of the last member (LenB without trailing padding)
        u32 nItems
        u16 tailPad      LenB - cbUnpadded
items   u16 plainBefore, u16 0, u8 op, u8 arg, u32 count, u16 extra      (12 bytes each)
        op: low nibble 0 String 1 Variant 2 Record 3 plain elements 5 padding 6 fixed s
           high nibble 0 scalar 1 fixed array (count = elements) 3 dynamic array (count =
        extra: fixed-string length / element size / offset from the item to its sub-layout
then    12 zero bytes per padding item that immediately precedes an array-of-records item
then    one sub-layout (same format) per record item; pad to 4 bytes
```

```

Type T: b As Byte: s As String: i As Integer: d As Double: x As String * 2: c As Currency: t

24 00 00 00 05 00 00 00 00 00          cbUnpadded=36, 5 items, no tail padding
01 00 00 00 05 03 01 00 00 00 00 00    1 plain byte (b), skip 3 padding
00 00 00 00 00 00 01 00 00 00 00 00    String s
02 00 00 00 05 02 01 00 00 00 00 00    2 plain bytes (i), skip 2 padding
08 00 00 00 06 00 01 00 00 00 02 00    8 plain bytes (d), String * 2 x
08 00 00 00 00 00 01 00 00 00 00 00    8 plain bytes (c), String t

file: 77 | 02 00 "AB" | 66 55 | ..f8 3f | "XY" | a8 61 .. | 02 00 "AB"

```

Figure 6. The Put/Get layout and the bytes `Put` wrote. Padding is skipped, fixed strings are written as ANSI characters, and Strings are written as a u16 length followed by their characters.

Plain members never get items; they accumulate into the next item's `plainBefore`. Every nested record, even a plain one, gets its own sub-layout, because its internal padding must be skipped as well. A dynamic array is written as a u16 dimension count, then (u32 count, u32 lbound) per dimension, then its elements. VB refuses at compile time to Get or Put a record containing an object reference, so there is no object operation.

12. Decoding algorithm

```

parse(p):
  cbDesc, cbRec, _, n, cbAnsi = u16 x5 ; align = u8 ; flags = u8
  repeat n:
    off = u16 ; t = u16 ; k = t & 0x1F
    if (t & 0x2000) or k in {05,09,0A,0B}: x = u16 ; ansiOff = u16
    if k == 0A: len = u16
    elif k == 09: pDesc = u32 ; skip 16 ; if pDesc: parse(pDesc)
    elif k == 05:
      et = u16
      if (et & 0x1F) == 09: skip 2 ; pDesc = u32 ; skip 16 ; parse(pDesc)
      elif t & 0x80: skip 2 ; p = u32 # fixed-string length, else 0
      elif (et & 0x1F) == 0A: skip 2 ; len = u16 # dynamic array of String * n
      if t & 0x80:
        cDims = u16 ; fFeatures = u16 ; cbElements = u32 ; skip 8
        bounds = cDims x (u32 cElements, i32 llbound) # rightmost dimension first
        if (et & 0x1F) != 09: trailer = u32 # VT, IID*, or 0x20
  assert bytes consumed == cbDesc

```

13. Open questions

- The meaning of the 0x20 trailer on fixed-length-string arrays.
- The 16 reserved bytes after each descriptor pointer. They may be an IRecordInfo cache filled at run time.
- The compiler-owned tail of module data blocks, and the 12 bytes following class member data.
- Entry types 0x06 and 0x08 and the 0x400 modifier, which the runtime handles but no probe has produced yet.

- Why the Put/Get layout reserves 12 zero bytes per padding item that precedes an array of records. The rule is exact; the reason is unknown.
- Still to probe: UDTs in Variants, and confirming that native-code builds (`__vbaRecAssign` and related) consume the same descriptors.

14. Reproducibility

The probe generator, decoder, independent encoder, scope and runtime checks are in the `pcode_db` project under `probes/_tools/udt_*.py`. The full regression set is `probes/_udt/udt_regression.json`. It holds 114 cases, each with VB source, raw descriptor bytes and the decoded tree, so other parsers can test against it.

The workflow used here (VB6 command-line compilation driven by an AI agent, with `vbdec` automation over the ROT) is described in the `vbdec` AI kit and the VB6 AI compile-help kit:

- sandsprite.com/vbdec/vbdec_ai.zip
- sandsprite.com/blogs/files/vbai_compile_help.zip